Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE KATAMIC MODEL OF TEMPORAL SEQUENCE PROCESSING: ANALYSIS AND MODIFICATIONS

J. B. Rosenberg

July 1993
CSD-930020

UNIVERSITY OF CALIFORNIA

Los Angeles

The KATAMIC Model of Temporal Sequence Processing: Analysis and Modifications

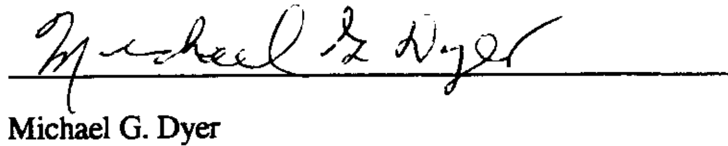
A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Jason Benedict Rosenberg

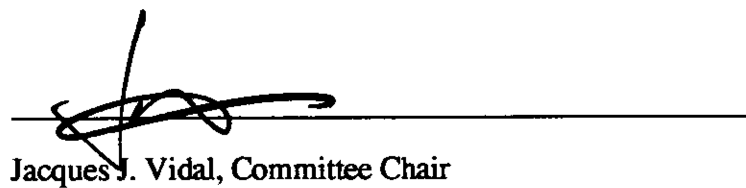
1993

The thesis of Jason Benedict Rosenberg is approved.



Michael G. Dyer

Allen Klinger



Jacques J. Vidal, Committee Chair

University of California, Los Angeles

1993

Table of Contents

Chapter One

Introduction 1

1.1 The Task 2

1.2 The KATAMIC Model 3

1.3 Understanding KATAMIC Learning 5

1.4 The Ksim Simulation Environment 6

Chapter Two

The KATAMIC Network Template 8

2.1 Network Structure 8

2.2 Neurons and Dendritic Compartments 9

2.3 Cycles, Learning, and the Flow of Information 10

2.4 Characterizing the Input and Output 12

2.5 Initializing the Network 14

2.6 Summary of Template Components and Parameters 14

2.6.1 Design Parameters--The Learning Module 14

2.6.2 Other Design Parameters 15

2.6.3 Configuration Parameters 15

2.6.4 Input Parameters 16

2.6.5 Some Default Settings 16

Chapter Three

Variations of the Learning Module 18

3.1 The Original KATAMIC Model 18

3.1.1 Specification of Loriginal 18

3.1.2 The Cycle Order 19

3.1.3 Removing The Spatial Decay Function 20

3.1.4 Benchmark Simulations 21
3.1.5 Increasing the Bit Density 25
3.2 Avoiding the Dependence on Sparse Input 26
3.2.1 Specification of LoriginalB 27
3.2.2 Simulations with LoriginalB 27
3.3 Simplifying the Computation Using Linear Update Rules 30
3.3.1 Specification of LoriginalC 30
3.3.2 Simulations with LoriginalC 30
3.4 The Bipolar Model 32
3.4.1 Specification of Lbipolar 33
3.4.2 Simulations with Lbipolar 33
3.5 Full Logic Bipolar Model 37
3.5.1 Simulations with LbipolarB 37
3.6 Perceptron Learning 40
3.7 Comparison of Learning Modules 42
3.7.1 Why the Bipolar Model Shows Better Convergence 46
Chapter Four
Some Further Simulations 47
4.1 Testing Convergence Relative to the Number of Inputs 47
4.2 Testing Convergence Relative to Dendritic Density 48
4.3 Testing Convergence with Duplicate Input Vectors 49
4.4 Testing Convergence with Noisy Input 50
4.5 Removing the ITV Wrap-Around During Shifting 50
Chapter Five
On Scalability and Hardware Implementation 52

Chapter Six

Concluding Remarks 54

6.1 The Virtues of Simplicity 54

6.2 De-Constructing the Neuroscience Perspective 55

Appendix A

Changes to KATAMIC Terminology 57

Appendix B

Overview Ksim 59

B.1 List of Commands for Ksim 59

B.1.1 Main Commands 59

B.1.2 Batch Commands 60

B.1.3 Configuration Commands 61

B.1.4 Input Commands 61

B.1.5 Learn Module Commands 62

B.1.6 Run Commands 63

B.1.7 Routing Commands 64

B.2 Template for Learning Module Header File 64

References 66

ACKNOWLEDGEMENTS

I would like to thank the following people for their help in completing this project:

My wife Roxanne, who served as chief editor. That she is now conversant in KATAMIC memories is testimony that it was a labor of love. She is my primary inspiration, and I shall return the favor by helping her to complete her dissertation.

My advisor, Jacques J. Vidal, for introducing me to the study of neural networks and guiding me in choosing the topic for this thesis.

Valeriy Nenov, who shared his work on the subject openly, and motivated me to learn all I can about neuroscience.

Michael McNally, who provided an important perspective for this work.

ABSTRACT OF THE THESIS

The KATAMIC Model
of Temporal Sequence Processing:
Analysis and Modifications

by

Jason Benedict Rosenberg
Master of Science in Computer Science
University of California, Los Angeles, 1993
Professor Jacques J. Vidal, Chair

Previous research has shown the KATAMIC neural network model to be quite suitable for temporal sequence recognition tasks. This thesis explores modifications to the model, in an attempt to improve its learning behavior. In addition, the original description of the KATAMIC framework has been simplified and formalized.

Through extensive experiments using the KATAMIC simulation environment Ksim, the present research demonstrates that learning capacity can be increased by a factor of six, while computational simplifications offer a speedup of 4 to 1 under simulation. The amount of storage required to implement the KATAMIC neuron can also be reduced by 33%.

These improvements are achieved by replacing the original dual valued long-term memory component with a single valued learned trace value (LTV). In addition, continuous valued dendritic trace components can be updated with a simple linear update

rule, as opposed to the more computationally expensive sigmoidal update function. Finally, the model has been altered to make it functional in the case of non-sparse input. This is accomplished by allowing the input to have bit values of -1 and 1, as opposed to the original 0 and 1.

Chapter One: Introduction

The KATAMIC memory model is a parallel neural network which has proven to be quite suitable for sequential, temporal pattern recognition tasks. Notably, the model has the ability to learn new pattern sequences quickly without a lengthy, repetitive training period. Its storage capacity compares favorably with other neural networks proposed to solve similar memory tasks.

The KATAMIC model was first proposed by Valeriy Nenov in 1990 [8].[1] In his Ph.D. dissertation, the model is fully described and employed as part of a larger project concerned with natural language acquisition [9]. Nenov was able to achieve some exciting results using KATAMIC memories for learning and storing verbal representations for visually perceived objects.

However, this first application of the model was part of a larger ambitious research project whose scope exceeded that of the KATAMIC model itself. Although Nenov analyzed some of the basic characteristics of KATAMIC memories, many interesting questions regarding the model were left unexplored. The present research is an attempt to further the understanding of the KATAMIC framework.

In this thesis, I have removed the KATAMIC model from the domain of its possible applications. Instead, I have concentrated only on the essential *learning* behavior of the model. Throughout, I have attempted chiefly to analyze the most general cases, without presuming specific structure within the input. By limiting the context in this way, it has been my aim to formalize the model more rigorously, and to better understand the conditions for its optimal performance.

I have experimented with several modifications to the model in attempts to simplify its structure and/or to improve its capacity and efficiency. I have conducted experiments

[1] The acronym 'KATAMIC' is derived from the names of Nenov's children, Katarina and Michael.

with varied parameter settings and network configurations, so as to compile strong empirical backing for the conclusions of this thesis. My results show that it is possible to speed up simulation of the model by a factor of four, while storage capacity can be increased by as much as six times.

I will also discuss the architectural simplicity of the KATAMIC model. I feel it is an attractive candidate for hardware implementation because it scales well, with non-exponential interconnection complexity as a function of network size.

The work for this thesis has engendered an integrated KATAMIC simulation environment called *Ksim*, which is outlined in section 1.4. All simulation results reported herein were obtained using Ksim.

1.1 The Task

The fundamental task for which KATAMIC memories are designed is the learning, recall and recognition of multiple pattern sequences of varying lengths. In the most studied case, the KATAMIC network is configured in a one-dimensional topology, such that the patterns comprising input sequences are binary-valued one-dimensional vectors. Two-dimensional networks have also been simulated, although I will not attempt to analyze that case at present [5,9]. The KATAMIC model is fully synchronous; a network receives input and generates output once during each cycle.

The basic KATAMIC task is analogous to that of the simple recurrent net (SRN) model [1,3]. Several comparative experiments were conducted which show the KATAMIC model to significantly out-perform the SRN model in terms of speed of convergence and in simplicity of computation per input pattern [9]. These tests showed that while SRN models indeed may be theoretically more equipped to perform any functional classification, the KATAMIC model offers quicker convergence on a wide range of useful input spaces. Where SRN may require hundreds of presentations of an input

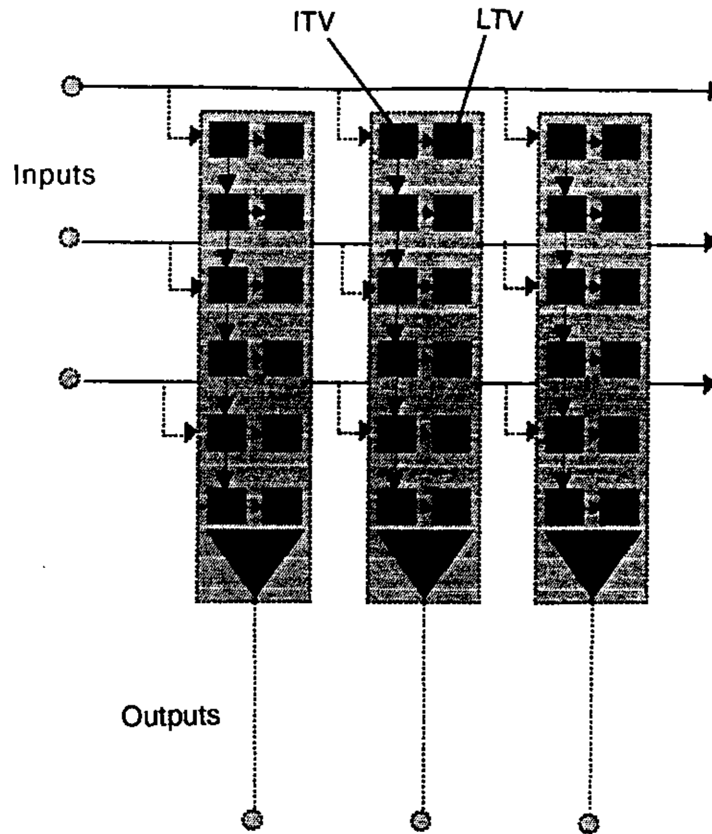


Figure 1.1 A simple KATAMIC network with three neural processing units.

corpus, the KATAMIC model will characteristically converge after fewer than 10 presentations.

1.2 The KATAMIC Model

In a KATAMIC network, the neural processing elements are relatively complex, when compared against many of the more mainstream neural network models popular among connectionists during the last decade [1, 2, 3, 4, 10]. Perhaps the most distinguishing feature of the KATAMIC neural network is its addition of processing structure within the input tree (dendrites) of each neuron. Although this substantially increases the computational cost of implementing or simulating each node, typical KATAMIC networks require far fewer neurons (and fewer repetition cycles) than do other models in performing comparable tasks.

Figure 1.1 shows a small KATAMIC memory. The KATAMIC network is composed of processing units which are viewed as simplified neurons. Each neuron has a series of *dendritic compartments*, which are linked in a chain. Each compartment maintains two basic types of data: an *input trace value* (ITV), and a *learned trace value* (LTV).[2] Temporal encoding is facilitated by shifting the ITV of each compartment one position after each input cycle (and allowing the value to decay).

Each bit of an input pattern gives rise to a *parallel fiber* which connects to one compartment of each neuron in the network. Since the KATAMIC neuron usually has more compartments than there are parallel fibers, only a subset of the dendritic compartments are connected to input. Those compartments which receive parallel fibers update their ITV during each cycle according to the input line to which they are connected. Compartments which do not receive direct input are *hidden compartments*.

The LTV is generally an adjustable weight which is updated by the value of its companion ITV during learning. The neuron's activation is typically determined by a comparison between its set of ITVs and LTVs throughout its dendritic tree. The binary output of the neuron is then a threshold function of its activation.

Once an output has been generated, it is compared against a teaching value (which can be simply the next value in the input corpus, or a supervised training set). Learning occurs only in those neurons which do not correctly match the expected output. The structure of the ITVs and LTVs is related to the choice of a learning update rule and the activation comparison function. These parameters concerned with learning comprise the *learning module*.

Nenov also employed additional processing units in his original description of the model that were concerned with recognition of the output and routing of the input (see

---

[2]My use of ITV and LTV is a change from Nenov's original short-term memory (STM) and long-term memory (LTM). There are several changes in terminology within my description of the model, which are summarized in Appendix A.

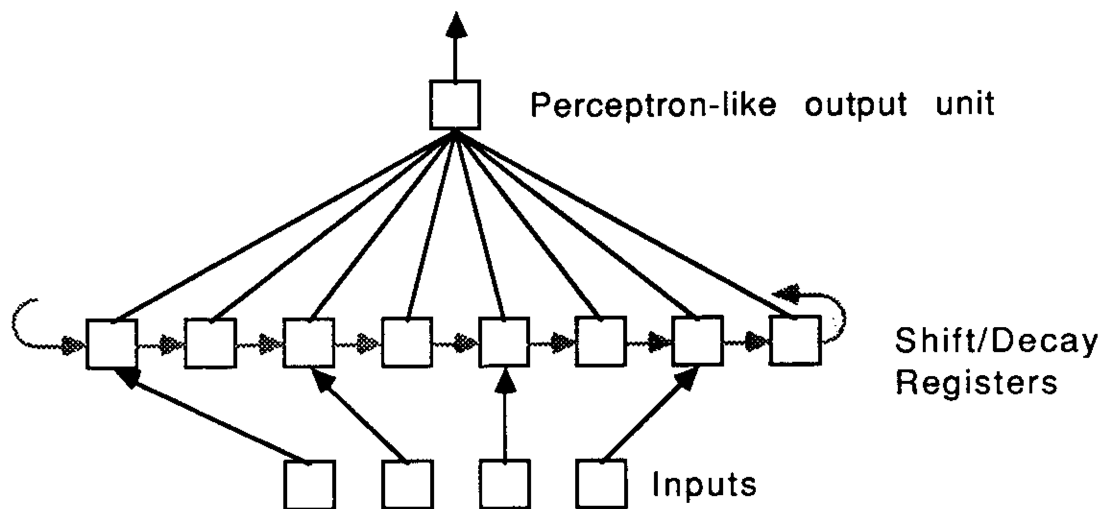


Figure 1.2 An alternate view of the bipolar KATAMIC neuron (courtesy of Michael McNally).

Appendix A). I have chosen not to incorporate these aspects of the model here, since the learning phenomena of interest occurs entirely in the dendritic trees of each neuron. The present research is aimed at studying this essential core of the model. However, my findings should be readily applicable to the original KATAMIC system.

1.3 Understanding KATAMIC Learning

It has been suggested that each KATAMIC neuron can be thought of as a simple perceptron node, with continuously valued connection weights (the LTV components) and inputs (the ITV components) [5]. This observation has gone a long way towards improving the understanding of the KATAMIC paradigm. Figure 1.2 shows a KATAMIC neuron viewed this way.

The ITV shifting mechanism serves to orthogonalize the set of input trace values over time, and thus the ITV becomes nicely compartmentalized into relatively dissimilar

vectors. This is the phenomenon thought to facilitate rapid learning in the model, since the spreading of the input serves to "jump-start" the job of distinguishing the different input states.

Overall storage capacity is aided by the addition of hidden compartments, which increases the size of the state space within which learning occurs. An input vector to the KATAMIC neuron is mapped over time to vector spaces of greater dimension. In Figure 1.2, this increase in vector length is a factor of two.

1.4 The Ksim Simulation Environment

Ksim is a full-purpose simulator for KATAMIC learning. It is the product of many months of development, refinement and experimentation. Written in *C++*, it has been designed in a modular way so as to allow for ease of implementing variations to the model [11]. Using the object-oriented features of *C++*, I have written five different KATAMIC *learning modules*, each of which interfaces identically to the Ksim simulation engine. Thus, it is convenient to experiment with different learning mechanisms given a fixed set of input conditions.

Ksim is built around a simple command interpreter. The user has the ability to configure the dimensions of the network to any arbitrary size, and has great flexibility in scheduling input and training vectors. Virtually any permutation of the model and simulation condition described herein can be accommodated using Ksim. Ksim also allows for inspecting the contents of relevant vectors and matrices at virtually any point during a cycle. It is also possible within Ksim to execute a series of commands using a batch file, in order to schedule a set of experiments, in a repetitive way. The state of the system can be saved to a file at any time, making it possible to repeat any test situation.

After running a prescribed number of cycles, Ksim always reports the percentage of correct outputs generated by the net over the current input corpus. It records the total time consumed and the time consumed per cycle, in both actual and CPU seconds. The CPU

time reported allows for the comparison of different variations of the model, according to computational efficiency (and implementation efficiency).

Appendix B contains a full accounting of Ksim's command set. Ksim runs under the UNIX operating system and is implemented sequentially. Unlike Nenov's simulations which were run on the parallel connection machine, Ksim is intended to allow for the focused study of varied parameters and conditions, which are more easily manipulated and maintained within the confines of a full purpose operating system such as UNIX. Ksim's execution speed is quite acceptable over a useful range of network parameters.

Empirical results reported in this thesis were achieved running Ksim on a Sun Microsystems 4/380 mini-computer, running SunOs 4.1.

Chapter Two: The KATAMIC Network Template

In this thesis, several versions of KATAMIC networks will be discussed, analyzed and simulated. As a starting point, it has been useful to define a base version of the model around which subsequent modifications can be described. It includes those features which are common to each KATAMIC variant.

This approach is of value since one of the goals of this thesis is to find simplified versions of the KATAMIC model which yield near optimal performance. The original description by Nenov is one of the more complex renditions of the model that will be explored herein. In presenting the basic network structure, I have attempted to retain only those features of Nenov's approach which are essential to the unique function and definition of KATAMIC memories.

In order to be general, this base version is delineated in terms allowing it to serve as a *template* for describing each variation to the model. This means that the descriptions for some of the network components are left under specified, although they are named and related to the overall structure. The set of features not fully characterized will be summarized. The description of each fully simulatable version of the model will completely specify each of these generalized features.

2.1 Network Structure

In referring to structural components and data objects throughout this thesis, I have used the convention that singular values are italicized, while matrices, sets and structured data are shown in bold italic symbols. Thus, the number of neurons in a network will be represented by N_n, while the representation for the x-th neuron and its set of component structures is referred to as $\boldsymbol{n}_x$.

Since this study is limited to one-dimensional network configurations, each neuron has two neighbors, except for those on either edge of the network which have just one

neighbor. It may be helpful to refer to the spatial representation shown in Figure 1.1 while reading what follows.

2.2 Neurons and Dendritic Compartments

As mentioned previously, each KATAMIC neuron is comprised by a set of dendritic compartments which participate in the calculation of an activation function; and each dendritic compartment has two types of data storage: information which reflects the input activity (ITV), and memory which is adjusted during learning (LTV). The neuron output is determined by an activation function A_f which generally performs a comparison of ITV and LTV values over the neuron's set of compartments.

The number of compartments per neuron N_c is generally uniform throughout a network; it is an adjustable parameter whose value can directly affect the network's storage capacity. The set of dendritic compartments $\boldsymbol{c_x}$ of a given neuron $\boldsymbol{n_x}$ is considered to be ordered, so that $\boldsymbol{c_x=\{c_{x,1},c_{x,2},\ldots,c_{x,Nc}\}}$. Likewise the sets of input trace and learned trace values for the neuron $\boldsymbol{n_x}$ are given by $\boldsymbol{ITV_x=\{ITV_{x,1},ITV_{x,2},\ldots,ITV_{x,Nc}\}}$ and $\boldsymbol{LTV_x=\{LTV_{x,1},LTV_{x,2},\ldots,LTV_{x,Nc}\}}$.

Within a network, a subset of the N_c dendritic levels have communicating parallel fibers. Each parallel fiber connects to every compartment within its level. The number of parallel fibers is given by N_p, where $N_p \leq N_c$. The ratio between N_c and N_p is the dendritic density $D_d=N_p/N_c$.

The subset of levels which contain parallel fibers can be characterized by the ordered distribution $\boldsymbol{k=\{k_1,k_2,\ldots,k_{Np}\}}$, where each k_i is unique and $1 \leq k_i \leq N_c$. The distribution of parallel fiber levels can be determined by a mapping function such that $\boldsymbol{k=K(N_p,N_c)}$.

Each node $\boldsymbol{n_s}$ can have one or more *seed* dendritic compartments; there is one seed per parallel fiber. At the seed, the transmission strength of a parallel fiber is at its maximum. The strength of a transmitted value then decreases as it spreads along the

parallel fiber in each direction, according to a decay function $d_d(\Delta_{s,y})$, where $\Delta_{s,y}$ is the distance between a distal node $\boldsymbol{n}_y$ and the seed node $\boldsymbol{n}_s$. It is important to note that although the signal drops off over distance, there are no temporal propagation delays along a parallel fiber; the entire network sees the value transmitted by a parallel fiber instantaneously.

Typically, fixed weights within dendritic compartments are used to implement the signal decay along a parallel fiber. In this case, the parallel fiber transmits at full strength throughout its extent, and each compartment c_{y,k_s} along a parallel fiber has its own input weight whose value is precisely $d_d(\Delta_{s,y})$.

In order to relate each parallel fiber to its seed neuron, it is useful to define the ordered set $s=\{s_1,s_2,\ldots,s_{Np}\}$, where $1 \leq s_i \leq N_n$ for each s_i. Thus, if the parallel fiber at level k_x has the neuron $\boldsymbol{n}_y$ as its seed, then $s_x=y$. The distribution of seed neurons can be given in general by the mapping function S, where $s=S(N_p,N_n)$. It is worth noting at this point that one of the conclusions of this thesis will be to call into question the need for seed compartments and the spatial decay along the parallel fibers (see Section 3.1.3).

2.3 Cycles, Learning, and the Flow of Information

The KATAMIC model is a fully synchronous neural network. Thus, the network's temporal behavior can be broken down into a series of time cycles, where each cycle represents a repeated operational sequence within the network. The system expects input and produces output once during each cycle. Following is a list of the essential operations for each cycle.

Input -- During each cycle t, an external synchronized input source generates the bit vector $i(t)$. In the simplest arrangement, each bit i_x of i connects directly to a unique parallel fiber at level k_x. Once the input has stabilized, each dendritic compartment c_{y,k_x} which receives a parallel fiber input modifies its ITV value according to an update function U_{ITV}, such that $ITV_{y,k_x}(t)=U_{ITV}(ITV_{y,k_x}(t-1),\ i_x(t)d_d(\Delta_{y,s_x}))$.

Output -- In this base version of the KATAMIC model, each bit o_x of the output vector $\boldsymbol{o}$ is taken directly from the output of each neuron $\boldsymbol{n}_x$, so that $o_x(t)=A_f(\boldsymbol{ITV}_x(t),\boldsymbol{LTV}_x(t))$.

Learn -- The output vector $\boldsymbol{o}(t)$ is compared against a training vector $\boldsymbol{t}(t)$; in the default case, the network is trained to predict the next input, so that $\boldsymbol{t}(t)=\boldsymbol{i}(t+1)$. Learning occurs in each neuron $\boldsymbol{n}_x$ for which $o_x(t) \neq t_x(t)$. During learning, each dendritic compartment of $\boldsymbol{n}_x$ updates its LTV component: $\boldsymbol{LTV}_{x,y}(t+1)=U_{LTV}(\boldsymbol{LTV}_{x,y}(t), \boldsymbol{ITV}_{x,y}(t), o_x(t)-t_x(t))$. Note that U_{LTV} should be constrained to produce no change in LTV if the third argument is 0 (i.e. $o_x(t)=t_x(t)$), since this not a learning condition).

Normalize -- After being updated, the set of LTV values for each neuron undergoing learning is normalized, such that the total activity within each dendritic tree remains constant or within prescribed bounds. We can represent this process formally with a function $N(\boldsymbol{LTV}_x)$. Normalizing is intended to assure that memory resources are adequately distributed and that the number of LTV values which are at a maximal value is limited within each neuron.

Shift -- Once during each cycle, the set of input trace values $\boldsymbol{ITV}_x$ within each neuron $\boldsymbol{n}_x$ is shifted one compartment in a uniform direction. So $\boldsymbol{ITV}_{x,y}(t+1)=d_s(\boldsymbol{ITV}_{x,y+1}(t))$, where d_s is a decay function which reduces each shifted ITV component. During shifting, the values may or may not wrap around from one end of the set of dendritic compartments to the other, such that $\boldsymbol{ITV}_{x,N_c}(t+1)=d_s(\boldsymbol{ITV}_{x,1}(t))$. Shifting is the mechanism which allows each neuron to learn a temporal context for a given input pattern.

The ordering of the operational steps within a cycle can be considered variable. Throughout this thesis, the following order will be used, unless otherwise stated:

1. Input (update ITV values from parallel fibers).
2. Output (generate the vector $\boldsymbol{o}(t)$).
3. Learn (update LTV values based on $\boldsymbol{o}(t)-\boldsymbol{t}(t)$).
4. Normalize LTV values.
5. Shift ITV values with wrap-around.

In order to be general, the ordered set F will be used to represent the sequence of operations for each cycle. Thus, in the above list we have *F={Input, Output, Learn, Norm, Shift}*. This formal representation allows for the straightforward specification of alternate cycle ordering, such as turning off learning and/or the normalization process (remove *Learn* and/or *Norm*), or changing the shift operation to one which does not wrap around (substitute *ShiftNoWrap* for *Shift*). In addition, as will be pointed out subsequently, Nenov's basic ordering varies slightly from the one presented above (see Section 3.1.2). The ordered list format for F mirrors the implementation of the cycle steps within Ksim.

2.4 Characterizing the Input and Output

The three bit vectors i, o and t contain the binary values 0 and 1 (or -1 and 1). The length of o and t is N_n, while the length of i is N_p. We can further characterize i by its bit density b and the redundancy ratios r_s and r_c. The bit density is the percentage of 1-bits in the input; it is used as a constraint in the generation of input vectors, so that each bit within i has a probability b of having the value 1. The redundancy ratios measure the percentage of duplicate vectors within a given sequence (in the case of r_s) and the percentage of duplicate vectors within a corpus (in the case of r_c). An easy way of generating an input corpus with a specified redundancy rate is to first generate it with all unique vectors, and then replace at random the appropriate number of vectors with vectors from elsewhere in the corpus. It is also useful to define an input noise ratio n, which corresponds to a percentage of input bits which have had their values flipped, relative to a previously learned corpus in which no noise was present.

KATAMIC input vectors are generally grouped into pattern *sequences*. A sequence of length L_s is expressed by $I=\{i_1,i_2,\ldots,i_{L_s}\}$. Furthermore, a set of L_c input sequences can comprise an input corpus $C_I=\{I_1,I_2,\ldots,I_{L_c}\}$, where the vector $L_s = \{l_1,l_2,\ldots,l_{L_c}\}$

represents the lengths of the constituents of C.[3] A training sequence $T=\{t_1,t_2,\ldots,t_{L_s}\}$ and a training corpus $C_T=\{T_1,T_2,\ldots,T_{L_c}\}$ must also be defined in conjunction with I and C_I.[4]

During each cycle t, the network generates the output vector $o(t)$ and compares it to the training vector $t(t)$. To characterize the network's convergence toward t, the percentage of correct bits in o is calculated. This is called the *correct* ratio. I will use this measure in lieu of the *match* ratio and *spurious* ratio measures defined and employed by Nenov.[5] In Nenov's simulations, he limited his input space to rather sparsely populated vectors, so that it was useful to measure convergence based solely on the correspondence of 1-bits between the output and teach vectors. I prefer to treat both logic levels as active components. As will be shown, it is possible to configure KATAMIC nets such that they exhibit acceptable convergence within the full-spectrum of input densities, not just in the sparse input case.

This is not to say Nenov's measures are not useful. Depending on the design of the learning module and the bit density setting, the convergence patterns based on the match and spurious measures can be different and are worth tracking independently. An alternate way to monitor this behavior is to define instead the two measures: *correct0* (the percentage of 0-bits in t which match o) and *correct1* (the percentage of 1-bits in t which match o). The choice to use either split measure scheme may be motivated by the intended application for a network. Perhaps there are uses for these networks for which it is only important to obtain good performance on one logic level. However, at present, I will not concern myself with such cases. Rather, I will study the conditions for the best convergence irrespective of logic levels. I feel this is necessary in order to define the KATAMIC model in the most general manner.

---

[3]The simulations reported here, however, will employ only uniform sequence lengths within a corpus. Thus, the single variable L_s will be used to represent sequence lengths. Implementation of variable length sequences is on the Ksim development to-do list.

[4]The separate definition of input and training sequences and corpi are not necessary in the case where $t(t)=i(t+1)$.

[5]These measures are defined in [9] as follows. The match ratio m is the number of correct 1-bits in o relative to the number of 1-bits in t. Likewise, spurious ratio s is the number of incorrect 1-bits in o relative to the number of 1-bits in t. The optimum value for m is 1 and that for s is 0.

2.5 Initializing the Network

Both the LTV and ITV values must be reset at the beginning of any network history; the structured constants LTV_{init} and ITV_{init} are assigned to each compartment during initialization. ITV values are also generally reset at the beginning of an input sequence, in order to clear any previous input traces.

Extra structure is added to the network in order to implement the initialization of the ITV in the form of a reset signal r_{ITV} which connects to every dendritic compartment in the network. In order to accommodate the generation of r_{ITV} at the beginning of an input sequence in a seamless manner, each input vector $i(t)$ within the Ksim implementation is grouped with the signal $r_{ITV}(t)$ as if it were a single vector of length N_p+1. Thus, it is trivial to schedule the resetting of input traces at any point during an input presentation corpus.

2.6 Summary of Template Components and Parameters

Following is an accounting of the various template parameters and structures which must be filled in for each working KATAMIC network. This list can be divided among those which are configuration parameters and those which are design parameters. Design parameters are needed for defining each version of the model, while configuration parameters are necessary for each instance of the model, regardless of version. Also summarized are parameters for characterizing the input. Note that parameters for the learning module are listed as a subset of the design parameters.

2.6.1 Design Parameters--The Learning Module

ITV -- The input trace memory structure, which resides in each dendritic compartment. This generally consists of a single real-numbered value. ITV_{init} is the initial condition for all ITV values in a network.

$U_{ITV}()$ -- The update function for each input trace value $ITV_{x,y}$ which resides in a compartment receiving parallel fiber input. It takes two arguments. The first argument is the previous value of $ITV_{x,y}$, while the second argument is the value carried by the parallel fiber at level y weighted by the decay function d_d. The output of U_{ITV} is the next value for $ITV_{x,y}$.

LTV -- The learned trace memory structure, which resides in each dendritic compartment. This generally consists of one or two real-numbered values. LTV_{init} is the initial condition for all LTV values in a network.

$U_{LTV}()$ -- The update function for each learned trace value $LTV_{x,y}$ which resides in each dendritic compartment. It takes three arguments. The first two are the current $LTV_{x,y}$ and $ITV_{x,y}$ values, while the third argument is the learning condition. The third argument can take on -1, 0 or 1. If it is 0, U_{LTV} returns an unchanged $LTV_{x,y}$. If it is -1 or 1, the neuron n_x produced an erroneous output and a learning condition exists; $LTV_{x,y}$ is updated in the appropriate direction.

$A_f()$ -- The activation function for each neuron n_x. It has two arguments, the first being the set of input trace values ITV_x for n_x, and the second being the corresponding set of learned trace values LTV_x. The output of A_f is a single binary valued bit.

$N()$ -- The normalization function for redistributing the learned trace values LTV_x for each neuron n_x after each learning step. The input and output of N is LTV_x.

$d_s()$ -- The decay function which reduces input trace memory while it is shifted at the end of each cycle. This function takes as input and outputs a single ITV value.

2.6.2 Other Design Parameters

$d_d()$ -- The decay function along a parallel fiber. It takes the distance between two neurons as its single argument, and outputs a real number value between 0 and 1.0.

$K()$ -- The mapping function for distributing the dendritic levels which contain parallel fibers. K takes two arguments, the number of neurons N_n and the number of compartments per neuron N_c. The output is the ordered set k.

$S()$ -- The mapping function for distributing the seed compartments for each parallel fiber. S takes two arguments, the number of parallel fibers N_p and the number of neurons N_n. The output is the ordered set s.

F -- The ordered set of operations to be performed on the network during each cycle.

2.6.3 Configuration Parameters

N_p -- The number of parallel fibers in the network; this determines the size of the input vector i.

N_n -- The number of neurons in the network; this determines the size of the two vectors o and t.

N_c -- The number of dendritic compartments per neuron. It is related to the number of parallel fibers and the dendritic density: $N_c = N_p/D_d$.

D_d -- The dendritic density. The ratio between the number of parallel fibers and the number of dendritic compartments per neuron. Thus, $D_d = N_p/N_c$.

$k = \{k_1, k_2, \ldots, k_{Np}\}$ -- The ordered set of dendritic levels for parallel fibers. This can be determined by the mapping function ***K***.

$s = \{s_1, s_2, \ldots, s_{Np}\}$ -- The ordered set of seed compartments for each parallel fiber. This can be determined by the mapping function ***S***.

2.6.4 Input Parameters

b -- The bit density for an input vector ***i***. It is the percentage of bits with value 1 within the vector.

r_s -- The vector redundancy within a sequence ***I***. It is the percentage of vectors which duplicate other vectors in the sequence. In calculating it, a duplicate pair is counted only once, so that if a sequence of length 10 has an r_s of 10%, then there will be 8 unique vectors, and 2 which duplicate each other. In other words, there will be a total of 9 different vectors in the sequence out of a possible 10.

r_c -- The vector redundancy within a corpus ***C***. Duplicates are not constrained to occur within the same sequence. It is calculated similarly to r_s, above.

n- -- The input noise probability. If noisy input is being simulated (i.e. $n>0$), then given an input corpus ***C***, each bit within ***C*** will be inverted with a probability n.

L_s -- The length of the input sequence ***I*** (and training sequence ***T***).

L_c -- The number of sequences in an input corpus C_I (and the training corpus C_t).

$\boldsymbol{L_s}$ -- The set of sequence lengths for the sequences which comprise C_I (and C_t).

2.6.5 Some Default Settings

Unless otherwise stated, very simple definitions for the mapping functions ***K*** and ***S*** will be used. ***S*** is defined such that each element of ***s*** is matched with the next consecutive neuron. That is, for each s_i within ***s***, $s_i = i \bmod N_n$. This function is called *inOrder*. ***K*** is defined in a similarly straightforward manner, using the *inOrderSpaced* function. In this scheme, ***k*** is a monotonically increasing ordered set of dendritic levels,

where each element is greater than the previous by a uniform interval. The interval is determined by the inverse of the dendritic density; so each $k_i=(integer)i/D_d$.

The input parameters r_s, r_c and n will be assumed to be 0, except in Sections 4.3 and 4.4, where they will be experimented with.

Chapter Three: Variations of the Learning Module

3.1 The Original KATAMIC Model

Nenov's original description of the KATAMIC model employs two learned trace values per compartment. One is updated only during learning conditions in which the desired output is positive (1), while the other is updated when the correct output is (0). To generate output, the Euclidean dot product between the ITV values and each set of LTV values is calculated; the output is determined by the LTV set which most closely correlates to the current ITV state. Sigmoidal update functions are employed for altering both the ITV and LTV values.

Following is the complete specification for the learning module as described by Nenov, stated in my terminology. I have tried to be faithful to the original version on all points. This version of the learning module will be referred to by its name within Ksim: *Loriginal.*

3.1.1 Specification of *Loriginal*

ITV -- A single, real-numbered value, ranging in the closed interval [0,1]. $ITV_{init}=0.001$, by default.

$U_{ITV}(A,B)=sig(sig^{-1}(A) + \varepsilon_{ITV}B)$, where *A* is a real numbered input trace value (i.e. $ITV_{y,kx}$), and *B* is the real numbered weighted input (i.e. $i_x(t)d_d(\Delta_{y,sx})$). The sigmoidal function is: $sig(x)=1-(1/(1+e^x))$ and the inverse is defined as: $sig^{-1}(y)=ln(1/(1-y)-1)$. The input update rate constant ε_{ITV} is an additional parameter. As a default, its value is set at *5.0.*

LTV -- Two real-numbered values, *pLTV* and *nLTV*, each of which can range in the closed interval [0,1]. LTV_{init} is the structured initial condition for each component. By default, it is set at *{0.5,0.5}*.

$U_{LTV}(A,B,C)$ is defined such that only one component of the learned trace value *A* (i.e. $LTV_{x,y}$) is updated, based upon *C*, the error direction (*-1, 0 or 1*). *B* is a real numbered input trace value (i.e. $ITV_{x,y}$). If *C* is 1, then $pLTV_{x,y}$ is updated as follows: $pLTV_{x,y} \leftarrow sig(sig^{-1}(pLTV_{x,y}) + \varepsilon_{LTV}B)$. If *C=-1*, $nLTV_{x,y}$ is updated similarly. No change in *A* occurs for *C=0.* The learning rate constant ε_{LTV} is an

additional parameter, whose default value is *1.0.* The sigmoid (and its inverse) are defined as for U_{ITV} above.

$A_f(A,B)=P(A \cdot B)$, where A is a vector of learned trace values (i.e. LTV_x), B is a vector of input trace values (i.e. ITV_x), $\cdot$ is the Euclidean inner product between A and the difference between the two components of B (i.e. $pLTV_x$-$nLTV_x$); P is the threshold function: $P(x)=1$ *iff* $x>\Theta_p$, *else* $P(x)=0$. Θ_p is set at 0 as a default.

$N(A)$ is defined such that the total amount of activity within each LTV component remains constant during learning. Thus, the value of LTV_{init} is used to set the amount of activity in the initial condition: $a_{init}= LTV_{init}*N_c$. For each $pLTV_x$ which has undergone learning, N updates each element $pLTV_{x,y}$ by multiplying it by a_{init}/a_x, where a_x is the sum of all elements in $pLTV_x$. The calculations are the same for updating $nLTV_x$.

$d_s(A)=Ae^{T_s}$, where A is an input trace value ($ITV_{x,y}$). The temporal decay constant T_s is set to -0.01 as a default.

3.1.2 The Cycle Order

One alteration I have made to the original description is in the cycle order. Nenov outlined a nine step cycle sequence, which can be condensed in my terms to *F'={input, learn, norm, shift, output}*. Notice that when there is an error in the output, learning will occur using input that did not exist when the erroneous output was generated. The ordering *F'* can be understood in part by noting that Nenov viewed the KATAMIC neuron, which he called a *predictron*, as a predictor of the next input (see Appendix A). Thus, the *output* step at the end of the cycle is essentially a prediction for the next input. But since the *input* step includes the updating of the ITV values in the network, the condition which generated the prediction is corrupted, and learning is impaired.

I have changed the order to the more straightforward *F={input, output, learn, norm, shift}* previously outlined. I am able to do so by relaxing the constraint that the KATAMIC neuron predict its next input; instead, a teach vector *t* is used, which may or may not contain a copy of the next input. If it does, I have made the plausible assumption that the next input will be available in the current cycle, to be assigned to *t*.

I have been unable to achieve acceptable learning using Nenov's original cycle order. I do not believe that this is due to my having left out much of Nenov's structure

concerned with recognition and feedback, since the essential learning still must occur within the dendritic structure of the neuron. However, it is conceivable that by alternately feeding back the internal output to the input, as can occur within Nenov's full model, the learning can be intermittently jarred out of stable local minima until a correct configuration is reached, as in simulated annealing. I have not attempted to analyze and simulate this case further, however.

3.1.3 Removing The Spatial Decay Function

Nenov defines the exponential spatial decay function $d_d(\Delta_{x,y})=e^{|\Delta_{x,y}|T_s}$, where $\Delta_{x,y}$ is the distance between two neurons n_x and n_y, and the spatial decay constant T_s is a variable parameter (default: $T_s = -0.01$).

The need for a spatial decay function d_d is unnecessary, however, in the absence of lateral interaction between neurons. It can be replaced by the constant function $d_d = 1$. Intuitively, this makes sense, since the spatial decay serves only to reduce the amount of context information available to each neuron. My investigations have shown this to be a valid simplification.

The use of input spatial decay may be advantageous in some instances, such as when statistical dependence exists between the input and training corpi. In this case, one or more input bits might correlate strongly with only a subset of the bits in the teaching corpus. Thus, one would want an input parallel fiber to have as its seed the neuron whose output needs to have the strongest correspondence with it. Implementation of a perceptive field is one example, since a neuron may only be concerned with a localized subset of the input arriving in its neighborhood. The spatial decay function would allow this neuron to observe distal activity to an exponentially decreasing degree.

However, the work in this thesis is primarily concerned with learning in the most general cases, in which there is no such correlation between input lines and output lines. As a consequence, no decay function will be used in subsequent simulations. This also

means that the notion of a dendritic seed, the seed distribution s, and the mapping function S are no longer needed. Arguably, any optimization of the learning module will remain valid in the presence of spatial decay.

Removing the spatial decay function essentially has the effect of rendering each neuron independent within a network, since relative position is no longer important. Likewise, the input trace values will be identical at all times for each neuron, as long as they receive the same input lines and have the same dendritic configurations and connectivity. This means that in simulation, at least, it is only necessary to update one set of ITV locations.

With this result in mind, the number of neurons in a network being simulated can be somewhat arbitrary. When testing for storage capacity or speed of convergence, for example, having multiple neurons reduces to simply having multiple test cases for a given input corpus. Each neuron must learn a different training sequence, but will receive the same input. Thus, running a network is somewhat like obtaining an averaged result. As a convention, I have run experiments with $N_n=5$ or 10.

3.1.4 Benchmark Simulations

As a benchmark simulation, I have repeated two important tests run by Nenov [9]. There are several motives for doing this. The first is for the sake of scientific completeness; it is important to duplicate results achieved by previous research. Successfully simulating the experiments also provides validation for the Ksim environment and supports the assumptions I have made in altering the original description. Finally, I wish to have a standard testbed in order to compare subsequent versions of the learning module.

The experiments duplicated are intended to test for storage capacity. In each case, a network with 64 inputs, 64 neurons and a dendritic density of 0.25 was used ($N_p=64$, $N_n=64$, $N_c=256$). In light of the conclusions of the previous section, I will use a network

with just 10 neurons. The first test attempts to measure the capacity in the case of a single input sequence ($L_c=1$). The network is trained on sequences of increasing length from 10 to 100 vectors, in increments of 10. The network is allowed a maximum of 40 cycles to learn each sequence. In this test, the input was constrained to a 1-bit density b of 15%. An input set for this test was generated, and will be used as input for subsequent versions of the model. For reference, this input set will be labeled: *LsTest_15* (i.e. sequence length test, with $b=0.15$). *LsTest_15* contains five different corpi for each sequence length increment, so that experiments with it can better demonstrate average learning behavior for a given sequence length. Unless otherwise stated, subsequent input sets will contain five input corpi for each parameter setting.

Table 3.1 shows the results of this experiment. The format of this table will be repeated for subsequent tests. The number of input presentations required by the network to completely learn each of the five input corpi is shown by each sequence length increment. In cases where the network was not able to obtain 100% convergence, the correct ratio after 40 repetitions is shown in parentheses. Also shown is the execution time consumed per KATAMIC cycle (in terms of CPU time) for the fifth trial of each sequence length.

Table 3.1 also highlights two performance milestones. The first is for *rapid convergence*, which will be defined as the longest sequence learnable by the network in ten or fewer repetitions. As a convention, this level will be demarcated by a light outline at the largest input level for which at least three trials meet the condition (i.e. $L_s=30$ in Table 3.1). Another milestone is for the number of sequences learnable after 40 repetitions. This will be referred to as *long convergence*. In the table format, the largest input level for which three or more trials exhibit long convergence will be denoted by shading (i.e. $L_s=40$ in Table 3.1).

The results in Table 3.1 only partially agree with those reported by Nenov. Nenov's experiment showed acceptable convergence for all tests where $L_s \leq 50$. His network also learned a sequence of length 60 to roughly 90% accuracy, before stabilizing. He used this case (L_s=60) in a calculation of the network's storage capacity.

In my experiment, learning is perfect for all sequences of length 10, 20 and 30. In these cases convergence is also quite rapid. For L_s=40, convergence is still quite successful. In four out of the five trials, the network learned the sequence, and in the fifth it learned the sequence to within 98% accuracy. For L_s=50, however, only one of the five trials exhibited near convergence with a correct ratio of 96%. No convergence was observed for any sequences with length greater than 50.

In comparing Nenov's results with Table 3.1, it is useful to point out that Nenov ran only one trial for each setting of L_s. As my results show, there can be a wide variance in the learning behavior for a given set of parameters. The results for sequence lengths 40 and 50 are good examples of this. It is conceivable that in conducting only one trial, the network could be seen to show good convergence for L_s=50. My numbers do not show any tendencies toward convergence for L_s=60, however.

Sequence Length Test with *Loriginal* and *b=0.15*						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	2	*3*	*4*	*5*	(*ms/cycle*)
10	4	3	4	4	4	48
20	5	6	6	6	5	45
30	6	7	6	5	5	44
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
50	(.66)	(.87)	(.77)	(.96)	(.66)	63
60	(.68)	(.39)	(.52)	(.48)	(.45)	80
70	(.56)	(.56)	(.60)	(.54)	(.60)	73
80	(.52)	(.54)	(.49)	(.56)	(.49)	76
90	(.53)	(.62)	(.53)	(.60)	(.60)	66
100	(.60)	(.42)	(.62)	(.43)	(.54)	72

Table 3.1 Convergence results for different single sequence lengths, using a network with the *Loriginal* learning module, 10 neurons, 64 inputs, dendritic density of 0.25, and input 1-bit density of 0.15. Tabulated under "trials" are the number of repetitions of the input needed to completely learn each training set (or the correct ratio reached after 40 repetitions, listed in parentheses). The right column lists the average cpu time required during each KATAMIC cycle for the fifth trial of each row. The outlined level represents the highest level at which three or more trials show *rapid convergence*, while the shaded row depicts the similar case for *long convergence*.

The cpu time listed in Table 3.1 is of interest. Notice that the number of milliseconds per KATAMIC cycle increases as the correct ratio decreases. This is perhaps an artifact of Ksim's sequential implementation of the model. Since each neuron only undergoes learning and normalization during cycles in which it produced an incorrect output, one Ksim optimization is to execute those steps only when they are needed. If Ksim were implemented in parallel, the execution time would be more constant for a given network configuration, regardless of input and convergence.

The second duplicated experiment measures the number of different sequences of length 10 that are learnable by the network. The network is allowed 40 cycles to train on input corpi of lengths 1 through 20. In this test, b is set to 10%. Once again, the input set used here is generated and set aside for use with subsequent experiments. It will be referred to as *LcTest_10*. Table 3.2 shows the results.

Multiple Sequence Test with ***Loriginal*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time (*ms/cycle*)
L_c	*1*	*2*	*3*	*4*	*5*	
1	4	4	3	4	3	51
2	4	5	4	3	4	41
3	4	4	6	4	6	40
4	4	5	7	5	5	40
5	5	5	5	5	4	43
6	5	7	7	9	6	38
7	7	6	5	6	7	40
8	7	6	6	6	6	39
9	6	14	6	5	7	42
10	9	6	8	8	6	40
11	6	9	6	7	14	39
12	8	11	10	17	12	39
13	7	15	10	6	8	39
14	12	10	17	14	12	38
15	18	31	(.996)	18	22	38
16	(.99)	14	15	21	26	37
17	(.99)	25	15	16	12	39
18	(.89)	25	30	(.99)	(.98)	39
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
20	(.99)	(.96)	(.94)	(.97)	(.91)	44

Table 3.2 Convergence results based on the number of different sequences of length 10, using a network with the *Loriginal* learning module, 10 neurons, 64 inputs, dendritic density of 0.25, and input 1-bit density of 0.10.

Nenov conducted this test with 5, 10 and 20 10-vector sequences. His network was able to learn 5 sequences quickly (after less than 5 presentations), and learned 10 sequences to 97% accuracy after 10 repetitions. However, he reported that for 20 sequences, the performance was "poor." My results show acceptable convergence for virtually all cases tested. Rapid convergence occurs quite reliably for $L_c \leq 11$. Notice that the cpu time is more constant, as opposed to the previous experiment. This is a result of improved convergence.

The results of this experiment show that the *Loriginal* learning module is better at learning multiple short sequences than it is at learning a single long sequence. As Nenov pointed out, this is due to the fact that ITV values are reset at the beginning of each sequence presentation. Thus, with multiple short sequences, the input trace is reset often and exhibits more sparsely populated activity patterns.

3.1.5 Increasing the Bit Density

Nenov's original version of the model is optimized to obtain good performance on sparse input vectors. This can be seen by observing the nature of the ITV update function, as defined in the *Loriginal* learning module. In this case, U_{ITV} is based on an active/passive logic scheme. That is, the ITV value will be adjusted only if the input is 1. If the input is 0, there is no change. As a result, the network is selectively sensitive to 1 bits. This is helpful for learning sparse input vectors, since only a few active ITVs are needed to uniquely identify a given input state. However, this strategy is not as useful when the bit density increases to 50%, since in this case 0-bits are equally important in distinguishing input states. When the bit density is very high, it is the 0-bits which contain the most interesting information.

To see how the *Loriginal* learning module performs under increased bit density, the two experiments of the previous section were repeated using $b{=}0.50$. The input sets *LsTest_50* and *LcTest_50* were generated. Tables 3.3 and 3.4 tabulate the results.

Sequence Length Test with ***Loriginal*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	6	5	4	5	5	47
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
30	(.80)	(.79)	(.84)	(.81)	(.81)	51
40	(.61)	(.60)	(.65)	(.58)	(.60)	66
50	(.53)	(.53)	(.57)	(.50)	(.60)	67
60	(.56)	(.47)	(.47)	(.45)	(.54)	69

Table 3.3 Results from sequence length test, as in Table 3.1, but with *b=0.50*.

Multiple Sequence Test with ***Loriginal*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	5	4	5	5	4	47
2	6	7	5	6	5	46
3	7	7	6	6	7	44
4	9	9	8	10	8	45
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
6	(.94)	28	(.94)	(.83)	(.86)	48
7	(.92)	32	(.81)	(.72)	(.82)	53
8	(.74)	(.73)	(.75)	(.71)	(.80)	56
9	(.69)	(.71)	(.72)	(.73)	(71)	60
10	(.74)	(.70)	(.70)	(.71)	(.80)	56

Table 3.4 Results from multiple sequence test, as in Table 3.2, but with *b=0.50*.

Notice that convergence is significantly impaired by the increased bit density. The network is able to reliably learn a sequence of length 20 (as opposed to 40 for *b=0.15*). Similarly, convergence drops below acceptable levels for more than 5 multiple sequences of length 10 (as opposed to 19 for *b=0.10*). In these tests, there is little distinction between rapid and long convergence. Performance seems to degrade quite abruptly after the boundary of rapid convergence is reached.

3.2 Avoiding the Dependence on Sparse Input

In order to achieve more successful learning for the case of non-sparse input, the update function U_{ITV} must be equally sensitive to 0s and 1s. This can be achieved by treating input 0-bits as if they were -1, and updating the ITV values either positively or negatively. Following is the specification for a modified version of *Loriginal* which does

this. The sigmoid functions for both the ITV and LTV values are shifted to range from -1 to 1 (as opposed to 0 to 1). The normalization function is altered as well. Since the range of the LTVs is now centered about an initial condition of 0, it is no longer sensible to constrain the net activity to remain constant, based on the initial conditions. Instead, an upper bounds for activity is defined, and normalization only occurs when the boundary is exceeded.

3.2.1 Specification of *LoriginalB*

ITV -- A single, real-numbered value, ranging in the closed interval [-1,1].[6] $ITV_{init}=0.0$, by default.

$U_{ITV}(A,B)=sig(sig^{-1}(A) + \varepsilon_{ITV}B)$, where A is a real numbered input trace value (i.e. $ITV_{x,y}$), and B is the input (i.e. $i_x(t)$). The sigmoidal function is changed to range from -1 to 1: $sig(x)=1-(2/(1+e^x))$ and its inverse is defined as: $sig^{-1}(y) = ln(2/(1-y) -1)$. The input update rate constant ε_{ITV} is set at 5.0 as a default.

LTV -- Two real-numbered values, ***pLTV*** and ***nLTV***, each of which can range in the closed interval [-1,1]. $LTV_{init}= \{0.0,0.0\}$, by default.

$U_{LTV}(A,B,C)$ Same as for *Loriginal*, except the sigmoid (and its inverse) are defined as for U_{ITV} above.

$A_f(A,B)=P(A \bullet B)$. Same as for *Loriginal*.

$N(A)$ is defined so that the total activity of the LTV of a given neuron remains within a bounds. Normalization only occurs if the activity surpasses the normalization threshold $\Theta_n=N_c*\Theta_{init}$. Θ_{init} is *0.50* by default. To normalize a set of $pLTV_x$ components, the activity a_x is calculated as the sum of the squares of each component $pLTV_{x,i}$, for $i=1,2,\ldots,N_c$. Each component $pLTV_{x,i}$ is multiplied by Θ_n/a_x if $a_x > \Theta n$. $nLTV_x$ is normalized similarly.

$d_s(A)=Ae^{Ts}$, same as for *Loriginal*.

3.2.2 Simulations with *LoriginalB*

The learning module *LoriginalB* is was tested on the previously generated sets *LsTest_50* and *LcTest_50*. The results are reported in Tables 3.5 and 3.6.

---

[6]Actually, within Ksim, this is currently implemented as ranging from [0,1], and U_{ITV} converts it to [-1,1] before proceeding. This was necessary in order to allow a common interface within Ksim and the different learning modules. The computational cost for this step is slight, as the cpu time reported in Table 3.5 shows (as compared to Table 3.1).

Sequence Length Test with ***LoriginalB*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	2	*3*	*4*	*5*	(*ms/cycle*)
10	4	4	3	4	5	46
20	4	5	5	4	4	48
30	6	5	6	4	4	50
40	5	5	5	5	5	46
50	6	7	6	5	7	44
60	6	6	8	8	8	44
70	9	6	9	7	8	45
80	9	9	9	8	8	46
90	12	8	10	10	9	44
100	12	10	10	16	16	43
110	18	12	22	12	15	42
120	20	17	21	21	14	42
140	(.996)	(.99)	(.999)	27	(.99)	41
150	(.96)	(.99)	(.96)	(.99)	(.98)	43
160	(.95)	(.92)	(.97)	(.98)	(.98)	44
170	(.90)	(.87)	(.93)	(.94)	(.93)	46

Table 3.5

Multiple Sequence Test with ***LoriginalB*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	2	*3*	*4*	*5*	(*ms/cycle*)
1	4	5	4	4	4	50
2	3	5	4	4	4	49
3	5	5	4	4	6	45
4	5	6	5	6	5	48
5	6	6	6	6	6	47
6	6	5	8	7	7	47
7	6	6	7	7	7	46
8	7	11	10	11	10	43
9	11	13	9	15	10	43
10	12	10	16	12	15	41
11	14	21	18	30	14	43
12	22	18	23	(.998)	35	40
13	34	22	30	27	18	43
15	(.99)	(.995)	(.94)	(.93)	(.99)	43
16	(.94)	(.94)	(.95)	(.98)	(.99)	43
17	(.94)	(.89)	(.89)	(.86)	(.85)	53
18	(.84)	(.90)	(.89)	(.87)	(.82)	54
19	(.86)	(.88)	(.80)	(.86)	(.85)	51

Table 3.6

As expected, *LoriginalB* significantly out-performs *Loriginal* when the input contains non-sparse vectors ($b=0.50$). In the sequence length test, it performs quite well, showing acceptable convergence for $L_s=130$, and rapid convergence for sequences of length 90. In the multiple sequence test, *LoriginalB* is successful with 14 sequences of length 10.

Notice that *LoriginalB* is as successful with single, long sequences as it is with multiple short sequences. This is an improvement over the behavior shown for *Loriginal*. The same phenomenon is most probably responsible. Remember that *LoriginalB* has been modified to be actively sensitive to both input logic levels, and thus is able to perform better in conditions of high activity within the input trace values.

It is important also to test *LoriginalB* in the sparse input case, as was done for *Loriginal*. Tables 3.7 and 3.8 show the results of input sets *LsTest_15* and *LcTest_10* run with *LoriginalB*. It is evident that *LoriginalB* is not successful with sparse input.

Sequence Length Test with ***LoriginalB*** and ***b=0.15***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time (*ms/cycle*)
L_s	*1*	*2*	*3*	*4*	*5*	
10	4	4	4	3	3	43
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
30	20	8	(.96)	(.95)	(.99)	37
40	(.89)	(.93)	(.94)	(.83)	(.94)	40
50	(.79)	(.74)	(.76)	(.80)	(.61)	55
60	(.62)	(.61)	(.67)	(.73)	(.56)	56

Table 3.7

Multiple Sequence Test with ***LoriginalB*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time (*ms/cycle*)
L_c	*1*	*2*	*3*	*4*	*5*	
1	4	4	4	4	5	41
2	3	3	4	5	4	41
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	(.96)	(.995)	(.98)	(.97)	(.95)	40
5	(.86)	(.87)	(.94)	(.92)	(.86)	51
6	(.94)	(.91)	(.75)	(.96)	(.93)	43
7	(.89)	(.81)	(.94)	(.64)	(.84)	48
8	(.89)	(.82)	(.68)	(.74)	(.74)	52
9	(.58)	(.67)	(.89)	(.77)	(.76)	50

Table 3.8

3.3 Simplifying the Computation Using Linear Update Rules

It is of interest to reduce the computational cost required to simulate the learning module. One way to do this is to replace the sigmoidal update rules used by U_{ITV} and U_{LTV} with simple linear update rules. This has been done with the *LoriginalC* learning module outlined below. One further change is to replace the exponential shift decay constant with a simple linear decay constant. This simplifies the specification nicely, although it does not offer any speedup since the decay function used in the exponential case can be pre-computed.

The specification for *LoriginalC* is identical to that for *LoriginalB*, except for those aspects listed.

3.3.1 Specification of *LoriginalC*

$U_{ITV}(A,B)=(A + \varepsilon_{ITV}B)$, where A is a real numbered input trace value (i.e. $ITV_{x,y}$), and B is the input (i.e. $i_x(t)$). The input update rate constant ε_{ITV} is set at 1.0 as a default.

$U_{LTV}(A,B,C)$ is defined such that only one component of the learned trace component A (i.e. $LTV_{x,y}$) is updated, based upon C, the error direction ($-1, 0$ or 1). B is a real numbered input trace value (i.e. $ITV_{x,y}$). If C is 1, then $pLTV_{x,y}$ is updated as follows: $pLTV_{x,y} \leftarrow (pLTV_{x,y})+\varepsilon_{LTV}B)$. If $C=-1$, $nLTV_{x,y}$ is updated similarly. No change in A occurs for $C=0$. The learning rate constant ε_{LTV} is an additional parameter, whose default value is 0.1.

$d_s(A)=\varepsilon_{dec}A$, where A is an input trace value ($ITV_{x,y}$). The temporal decay constant ε_{dec} is set to 0.5 as a default.

3.3.2 Simulations with *LoriginalC*

LoriginalC was tested on the four input sets *LsTest_15*, *LcTest_10*, *LsTest`_50*, and *LcTest_50*. The results are presented in Tables 3.9 through 3.12. As expected, the non-sigmoidal version executes significantly faster under Ksim than does *LoriginalB* (speedup of *~44/25=1.8*), but shows a slight degradation in convergence behavior. Like *LoriginalB*, *LoriginalC* is more successful when *b=0.50*.

Sequence Length Test with ***LoriginalC*** and ***b=0.15***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	6	4	5	4	4	25
20	8	8	7	6	5	25
30	9	6	6	7	8	24
40	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
50	(.78)	(.76)	(.85)	(.84)	(.84)	26
60	(.73)	(.84)	(.77)	(.83)	(.68)	29
70	(.73)	(.41)	(.51)	(.67)	(.49)	31

Table 3.9

Multiple Sequence Test with ***LoriginalC*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	7	5	3	4	6	25
2	5	5	5	7	6	24
3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	(.97)	(.98)	(.93)	(.93)	(.96)	24
5	(.89)	(.95)	(.88)	(.90)	(.86)	25
6	(.95)	(.91)	(.90)	(.93)	(.91)	24
7	(.86)	(.75)	(.87)	(.64)	(.91)	24

Table 3.10

Sequence Length Test with ***LoriginalC*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	4	3	4	4	3	26
20	4	4	5	4	4	25
30	6	5	6	5	5	25
40	6	5	6	6	7	25
50	10	7	8	7	7	26
60	9	9	8	8	7	25
70	8	11	8	9	8	27
80	11	10	9	8	11	25
90	10	10	12	12	13	25
100	19	14	15	14	14	25
110	14	20	20	13	13	25
120	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
130	(.98)	(.95)	(.98)	25	(.94)	25
140	(.85)	(.95)	(.90)	(.95)	(.91)	25
150	(.78)	(.86)	(.81)	(.82)	(.84)	26

Table 3.11

Multiple Sequence Test with *LoriginalC* and *b=0.50*						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	4	4	3	3	4	26
2	5	5	4	5	3	26
3	5	6	5	5	5	26
4	6	6	6	6	6	26
5	8	5	7	7	8	25
6	9	10	7	9	9	26
7	9	11	9	12	10	26
8	12	11	9	11	12	26
9	12	14	17	9	10	26
10	14	17	12	16	12	25
11	14	16	16	39	18	25
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
13	(.92)	(.94)	20	(.89)	(.96)	25
14	(.82)	(.76)	(.82)	(.68)	(.70)	28
15	(.76)	(.75)	(.77)	(.82)	(.77)	27

Table 3.12

3.4 The Bipolar Model

Michael McNally has demonstrated that it is possible to design the learning module with a single valued *bipolar* LTV component [5]. This is obviously an attractive improvement, since it decreases the amount of computation required in calculating the activation function A_f by a factor of two. It also reduces the amount of storage space required per dendritic compartment from 3 values to 2 (1 ITV component and 1 LTV component).

McNally did not observe any benefits for learning and convergence with his implementation of the bipolar modification. As my results will show, however, using only a single LTV component actually increases convergence and storage capacity. I have defined two bipolar versions of the learning module, *Lbipolar* and *LbipolarB*. The first uses an ITV update rule similar to *Loriginal* in that it is passive/active relative to input 0s and 1s, although it is not sigmoidal. *LbipolarB*, on the other hand, uses the full logic version of U_{ITV}, as in *LoriginalC*.

3.4.1 Specification of *Lbipolar*

ITV -- A single, real-numbered value. $ITV_{init}=0.0$, by default.

$U_{ITV}(A,B)=(A + \varepsilon_{ITV}B)$, where A is a real numbered input trace value (i.e. $ITV_{x,y}$), and B is the input (i.e. $i_x(t)$). The input update rate constant ε_{ITV} is set at 1.0 as a default. Note that U_{ITV} does not cause a change in value for $i_x(t)=0$.

LTV -- A single, real-numbered value. $LTV_{init}=0.0$, by default.

$U_{LTV}(A,B,C) = A + \varepsilon_{LTV}BC$, where A is a learned trace component (i.e. $LTV_{x,y}$), B is an input trace value (i.e. $ITV_{x,y}$), and C is the error direction (-1, 0 or 1). The learning rate constant ε_{LTV} has a default value of 0.1.

$A_f(A,B)=P(A \bullet B)$, where A is a vector of learned trace values (i.e. LTV_x), B is a vector of input trace values (i.e. ITV_x), $\bullet$ is the Euclidean inner product between A and B; P is the threshold function: $P(x)=1$ *iff* $x>\Theta_p$, *else* $P(x)=0$. Θ_p is a set at 0 as a default.

$N(A)$ is defined so that the total activity of the LTV of a given neuron remains within a bounds. Normalization only occurs if the activity surpasses the normalization threshold $\Theta_n=Nc^*\Theta_{init}$. Θ_{init} is 0.50 by default. The activity a_x is calculated as the sum of the squares of each component $LTV_{x,i}$, for $i=1,2,\ldots,N_c$. Each component $LTV_{x,i}$ is multiplied by Θ_n/a_x if $a_x>\Theta_n$.

$d_s(A)=\varepsilon_{dec}A$, where A is an input trace value ($ITV_{x,y}$). The temporal decay constant ε_{dec} is set to 0.5 as a default.

3.4.2 Simulations with *Lbipolar*

The results for *Lbipolar* running the input sets *LsTest_15* and *LcTest_10* are shown in Tables 3.13 and 3.14. As expected, simulating *Lbipolar* is significantly faster than any of the previous versions based on 2 LTV components per compartment (speedup of ~4 over *Loriginal*)

Notice that *Lbipolar* is superior to *Loriginal* in several ways, including execution speed, simplicity of design and, most importantly, learning behavior. It not only exhibits a relatively high storage capacity, but degrades in performance quite gracefully. In the multiple sequence test, it shows reliable long convergence for 31 sequences, but is able to successfully learn sequences of length 40 (or more) to 99% accuracy.

Sequence Length Test with ***Lbipolar*** and ***b=0.15***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	4	3	4	3	3	12
20	5	5	4	4	5	12
30	5	4	5	5	7	11
40	6	5	5	5	6	11
50	6	8	8	8	7	11
60	7	7	9	8	8	11
70	8	10	9	8	8	11
80	11	8	8	7	9	11
90	12	10	10	10	10	12
100	10	11	11	11	10	11
110	11	12	10	11	9	11
120	12	10	8	10	11	11
130	12	13	11	15	16	11
140	17	13	15	18	16	10
150	17	13	14	20	18	11
160	14	20	14	18	17	11
170	18	19	19	17	20	10
180	29	25	29	23	19	10
190	19	22	19	21	21	11
200	22	25	24	21	23	11
210	25	18	21	29	31	10
220	28	28	23	26	29	10
230	24	25	33	31	27	11
240	37	(.999)	32	29	38	10
250	32	34	38	(.999)	(.998)	10
260	38	(.998)	33	31	34	11
270	37	38	(.996)	(.997)	40	11
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
290	(.997)	(.998)	(.998)	(.99)	(.998)	11
300	(.996)	(.997)	(.998)	(.998)	(.99)	12

Table 3.13

Multiple Sequence Test with ***Lbipolar*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	4	4	3	3	3	11
2	4	4	3	3	3	11
3	4	3	4	5	5	11
4	5	4	6	4	8	11
5	5	6	4	6	6	11
6	5	6	8	6	8	11
7	7	8	6	7	8	11
8	7	11	7	8	5	10
9	10	8	6	7	6	11
10	10	7	7	9	11	10
11	7	8	8	8	10	10
12	10	11	11	7	16	10
13	12	10	12	14	12	11
14	10	13	9	11	15	10
15	17	13	15	12	16	10
16	22	15	14	13	18	10
17	11	20	13	15	10	10
18	22	13	23	17	20	10
19	12	14	17	19	13	10
20	16	20	20	25	(.999)	10
21	11	30	20	14	21	11
22	20	15	16	14	21	11
23	21	27	29	28	25	11
24	30	25	28	26	33	11
25	31	17	37	38	22	11
26	20	32	37	20	32	11
27	29	31	29	20	(.996)	11
28	25	(.997)	40	39	35	11
29	(.99)	(.99)	33	(.99)	34	11
30	(.998)	31	(.99)	(.999)	(.999)	11
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
32	(.998)	(.997)	(.99)	(.99)	36	11
33	24	(.997)	(.999)	(.999)	(.995)	11
34	(.99)	(.99)	(.99)	(.999)	28	10
35	(.997)	(.999)	(.99)	(.998)	(.998)	10
36	(.996)	(.996)	36	(.99)	(.996)	10
37	34	(.998)	(.997)	(.99)	37	10
38	(.99)	(.99)	(.98)	(.99)	(.998)	10
39	(.996)	(.99)	(.98)	(.99)	(.998)	10
40	34	(.97)	(.99)	(.98)	(.997)	10

Table 3.14

Sequence Length Test with ***Lbipolar*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	6	5	5	5	6	12
20	8	8	7	12	9	12
30	12	11	12	10	12	11
40	10	16	15	11	12	11
50	20	17	13	19	25	11
60	20	22	19	20	18	11
70	23	28	21	20	25	11
80	34	28	35	29	23	11
90	33	27	23	27	38	11
100	36	35	33	(.996)	32	11
110	33	(.99)	(.997)	32	39	11
120	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
130	(.99)	(.99)	36	(.997)	(.995)	12
140	(.99)	(.99)	(.99)	(.99)	(.99)	12
150	(.97)	(.98)	(.98)	(.99)	(.99)	12
160	(.98)	(.92)	(.92)	(.99)	(.98)	12

Table 3.15

Multiple Sequence Test with ***Lbipolar*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	6	5	6	7	6	12
2	6	7	6	7	6	13
7	7	9	9	8	8	12
4	8	14	8	11	15	11
5	8	7	12	10	10	12
6	14	11	10	12	9	13
7	14	12	17	14	21	11
8	12	11	15	15	16	11
9	17	19	25	15	15	12
10	12	15	17	14	14	12
11	17	18	25	22	20	11
12	29	19	23	25	30	11
13	27	26	21	30	27	12
14	21	32	31	26	26	11
15	29	24	(.996)	(.997)	22	11
16	35	37	29	38	28	11
17	28	26	34	32	33	12
18	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
19	(.999)	(.998)	(.99)	(.96)	36	12
20	(.99)	(.97)	(.99)	(.996)	36	12
21	(.999)	(.98)	35	(.99)	36	11
22	(.94)	(.998)	(.997)	(.95)	(.997)	11
23	(.94)	(.96)	(.95)	(.97)	(.98)	11
24	40	(.96)	(.89)	(.88)	(.96)	12

Table 3.16

Tables 3.15 and 3.16 show the convergence of *Lbipolar* for the non-sparse input sets *LsTest_50* and *LcTest_50.* As expected, there is a noticeable drop-off in performance for *b=0.50.* However, it is comparable with *LoriginalB*, for which non-sparse input is its best case.

3.5 Full Logic Bipolar Model

Although *Lbipolar* seems to have performed relatively well on the non-sparse input set, it is worth defining and simulating a full-logic version, which is actively sensitive to both input bit levels. The specification for *LbipolarB*, is the same as for *Lbipolar* in every regard, except for the input trace update function, which expects its second argument *B* to have values of *-1* and *1* instead of *0* and *1*.

3.5.1 Simulations with *LbipolarB*

The results for *LbipolarB* running the input sets *LsTest_15* and *LcTest_10* are shown in Tables 3.17 and 3.18. Predictably, *LbipolarB* shows a marked improvement over *Lbipolar* in the non-sparse input case. Tables 3.19 and 3.20 display the results for *LbipolarB* run on *LsTest_50* and *LcTest_50.*

Sequence Length Test with ***LbipolarB*** and ***b=0.15***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	2	*3*	*4*	*5*	(*ms/cycle*)
10	6	4	5	4	4	12
20	8	8	7	6	5	11
30	9	6	6	7	8	11
40	9	7	6	8	8	11
50	9	16	10	11	12	11
60	12	13	16	11	13	11
70	14	19	20	9	13	11
80	15	20	14	16	16	11
90	12	18	19	17	20	11
100	20	18	24	16	17	11
110	18	28	20	21	18	11
120	21	18	24	26	23	11
130	40	27	22	25	29	11
140	30	(.999)	35	25	33	11
150	(.999)	28	30	(.98)	(.98)	11
160	36	25	24	23	39	11
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
180	(.98)	35	(.97)	(.98)	(.995)	11
190	(.99)	(.97)	(.999)	(.995)	(.98)	11
200	(.99)	(.98)	(.97)	(.98)	(.98)	11

Table 3.17

Multiple Sequence Test with ***LbipolarB*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	2	*3*	*4*	*5*	(*ms/cycle*)
1	7	5	3	4	6	11
2	5	5	5	7	6	11
3	6	5	5	6	7	11
4	8	9	6	9	6	11
5	8	7	6	7	11	11
6	9	9	8	6	8	11
7	11	9	8	14	13	10
8	10	19	8	7	8	12
9	16	10	6	7	9	11
10	8	11	10	12	17	11
11	6	13	12	12	13	11
12	15	20	25	10	16	11
13	12	15	19	25	15	10
14	11	13	15	15	13	11
15	15	14	12	16	11	11
16	18	19	13	14	20	11
17	18	(.96)	26	17	16	10
18	(.98)	13	(.97)	(.97)	(.98)	10
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
20	(.96)	(.93)	(.98)	(.98)	(.97)	11

Table 3.18

Sequence Length Test with ***LbipolarB*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	4	3	4	4	3	14
20	4	4	5	4	4	14
30	6	5	6	5	5	13
40	6	5	6	6	7	12
50	10	7	8	7	7	12
60	9	9	8	8	7	12
70	8	11	8	9	8	12
80	11	10	9	8	11	12
90	10	10	12	12	13	11
100	19	14	15	14	14	12
110	14	20	20	13	13	12
120	13	15	16	14	17	11
130	18	17	20	16	14	12
140	23	20	25	18	24	12
150	28	21	23	21	25	12
160	29	30	27	31	24	12
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
180	(.96)	(.97)	(.92)	(.96)	(.89)	11
190	(.91)	(.90)	(.92)	(.82)	(.93)	12

Table 3.19

Multiple Sequence Test with ***Lbipolar*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	4	4	3	3	4	13
2	5	5	4	5	3	14
3	5	6	5	5	5	12
4	6	6	6	6	6	12
5	8	5	7	7	8	12
6	9	10	7	9	9	12
7	9	11	9	12	10	12
8	12	11	9	11	12	12
9	12	14	17	9	10	12
10	14	17	12	16	12	12
11	14	16	16	21	18	11
12	17	15	19	17	22	12
13	21	21	20	22	20	12
14	22	24	28	28	27	12
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
16	27	28	(.97)	(.98)	(.97)	11
17	29	(.94)	(.98)	32	(.86)	13
18	(.76)	(.83)	(.79)	(.79)	(.88)	14

Table 3.20

3.6 Perceptron Learning

In order to add perspective to the learning modules discussed so far, I have also run a simple *perceptron* version of the learning module on the same 4 input sets. It is implemented in Ksim as is *LbipolarB*, except that the ITV update function has no memory, so that $U_{ITV}=i(t)$. Also, shifting of the ITV is turned off, and the dendritic density is fixed at 1 (no hidden compartments). This learning module will not be a considered a variation of the KATAMIC model.

In the input sets tested, there are no redundant vectors (i.e. r_s and $r_c=0$). Thus, it is important to compare the KATAMIC models against an implementation which makes no use of temporal information and therefore can only learn on a one-to-one mapping between each input and teaching vector. The performance of the *Lperceptron* learning module will be viewed as a sort of minimum level for acceptable learning characteristics among the KATAMIC learning modules, which attempt to take advantage of temporal context. Tables 3.21 through 3.24 display the results for *Lperceptron.*

As expected, this learning module shows essentially the same performance for both the sequence length test and the multiple short sequence test. This is because it has no input trace values to be reset at the beginning of a cycle, and thus it is imperceptive to any sequential structure within an input corpus. *Lperceptron* is superior in learning behavior to both *Loriginal* and *LoriginalB* for some cases. Notice, however, that it does win big in the areas of simplicity and speed of execution.

Sequence Length Test with ***Lperceptron*** and ***b=0.15***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	5	4	4	4	3	3
20	7	9	9	8	6	3
30	6	7	7	6	9	3
40	11	11	8	14	9	3
50	19	27	13	14	18	3
60	23	13	17	17	18	3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
80	(.94)	(.95)	(.99)	21	(.96)	3
90	(.98)	(.99)	(.95)	(.97)	(.98)	3
100	(.97)	(.97)	(.92)	(.93)	(.97)	3

Table 3.21

Multiple Sequence Test with ***Lperceptron*** and ***b=0.10***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	5	7	3	5	7	3
2	7	6	10	7	5	3
3	5	5	7	7	10	3
4	13	17	20	7	12	3
5	13	17	13	10	(.94)	3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
7	22	(.98)	27	(.96)	(.97)	3
8	(.98)	(.95)	(.96)	25	(.98)	3
9	(.90)	(.97)	13	(.97)	(.97)	3
10	(.93)	(.98)	(.98)	(.90)	(.90)	3
11	17	(.96)	(.996)	(.97)	(.92)	3
12	(.95)	(.93)	(.92)	(.97)	(.90)	3

Table 3.22

Sequence Length Test with ***Lperceptron*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_s	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
10	4	3	3	4	3	3
20	5	5	5	5	5	3
30	8	5	5	5	6	3
40	7	9	8	9	15	3
50	12	10	12	13	11	3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
70	31	14	(.96)	(.96)	(.99)	3
80	(.87)	(.75)	(.81)	(.83)	(.89)	3

Table 3.23

Multiple Sequence Test with ***Lperceptron*** and ***b=0.50***						
	Repetitions to Convergence (*or correct ratio after 40 reps*)					Trial 5 cpu time
L_c	*1*	*2*	*3*	*4*	*5*	(*ms/cycle*)
1	4	5	4	3	4	4
2	5	6	4	4	4	3
3	6	6	7	6	5	3
4	8	8	9	10	7	3
5	14	8	13	9	12	3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
7	(.96)	34	(.98)	(.96)	(.92)	3
8	(.87)	(.89)	(.82)	(.73)	(.88)	3

Table 3.24

3.7 Comparison of Learning Modules

Table 3.25 summarizes the six learning modules tested in terms of execution efficiency and storage capacity. Of the KATAMIC variants, both bipolar versions offer a speedup of 4 over the original version within the Ksim environment, and use 33% less storage. *Lperceptron*, of course, is the fastest and least memory expensive model.

The learning behavior of the six learning modules over the four input sets tested is

Learning module	Typical cpu time (*ms/cycle*)	Memory locations used per neuron
Loriginal	44	768
LoriginalB	43	768
LoriginalC	25	768
Lbipolar	11	512
LbipolarB	11	512
Lperceptron	3	64

Table 3.25 Relative requirements for the six learning modules tested in terms of execution time and memory cost.

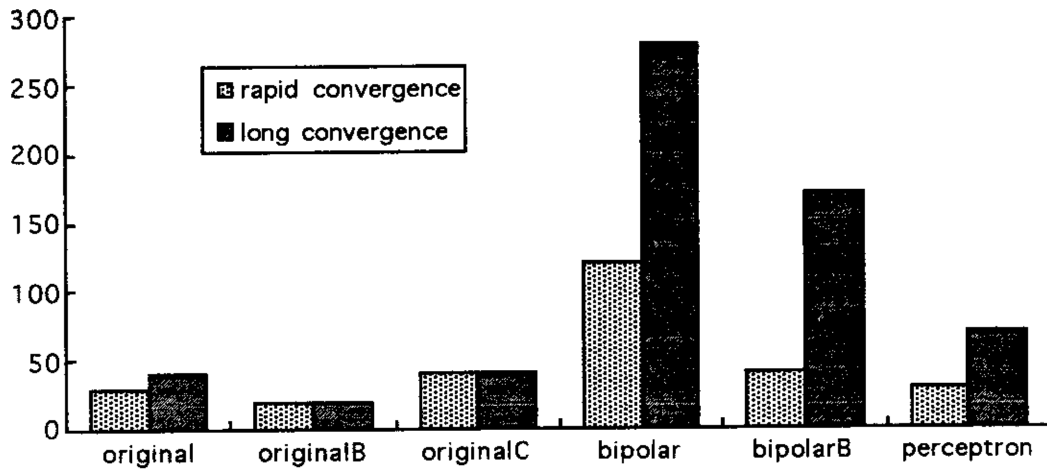


Figure 3.1 Relative convergence for six learning modules on the input set *LsTest_15*. The vertical axis represents sequence length.

displayed graphically in Figures 3.1 through 3.4. Both rapid convergence and long convergence are depicted.

Figure 3.1 shows the relative behavior for the input set *LsTest_15*. In this case, *Lbipolar* clearly has the best convergence characteristics. *LbipolarB* also does relatively well on this input set for long convergence, but is only average in the rapid convergence category. *Lperceptron* is superior to all variations of the original model in terms of long

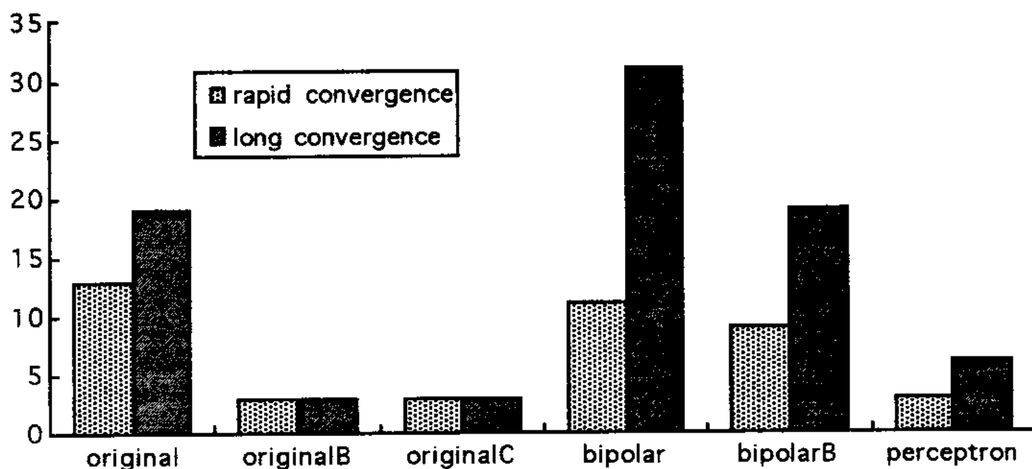


Figure 3.2 Relative convergence for six learning modules on the input set *LcTest_10* The vertical axis represents the number of sequences of length 10.

convergence.

In Figure 3.2, the same graph is displayed for the results with the input set *LcTest_10.* Once again, *Lbipolar* is superior for long convergence. But, both *Loriginal* and *LbipolarB* are successful, and *Loriginal* has the best rapid learning characteristics. Figures 3.1 and 3.2 reassert that *Loriginal* is better on multiple, short sequences. Notice that neither *LoriginalB* nor *LoriginalC* perform well in the sparse input case for multiple sequences, and are outperformed by *Lperceptron.*

The results for the non-sparse input set *LsTest_50* are depicted in Figure 3.3. *LbipolarB* is superior in terms of long convergence, while *LoriginalB* has a slight edge in terms of rapid convergence. Notice that *Lbipolar* has rather poor rapid convergence characteristics (worse than *Lperceptron*), but is relatively successful for long convergence. Given its superior execution speed and reduced storage requirements, *LbipolarB* is the clear winner in this test.

The comparative results for the fourth input set *LcTest_50* appear in Figure 3.4. There is no clearly superior candidate in the test, in terms of either convergence measure. However, the bipolar entries are better choices than *LoriginalB* and *LoriginalC*, since they not only have better long convergence, but also are also simpler and more efficient.

In these convergence graphs, the scale for sparse input is almost double that for the non-sparse case. This can be understood by observing that the sparse input learning task is a substantially easier problem, since the number of permutations of the input where b is low is far smaller than for $b=0.50$. This means that the number of possible classifications of the input is reduced substantially.

In these experiments, I have not reported the case where b is high (i.e. *0.90*). For those learning modules which are equally attentive to both input levels, the high bit-density case is statistically identical to the sparse input case. Therefore similar convergence characteristics can be assumed for *LoriginalB, LoriginalC, LbipolarB* and *Lperceptron* for high 1-bit density as was observed for sparse input. For *Loriginal* and *Lbipolar*, which

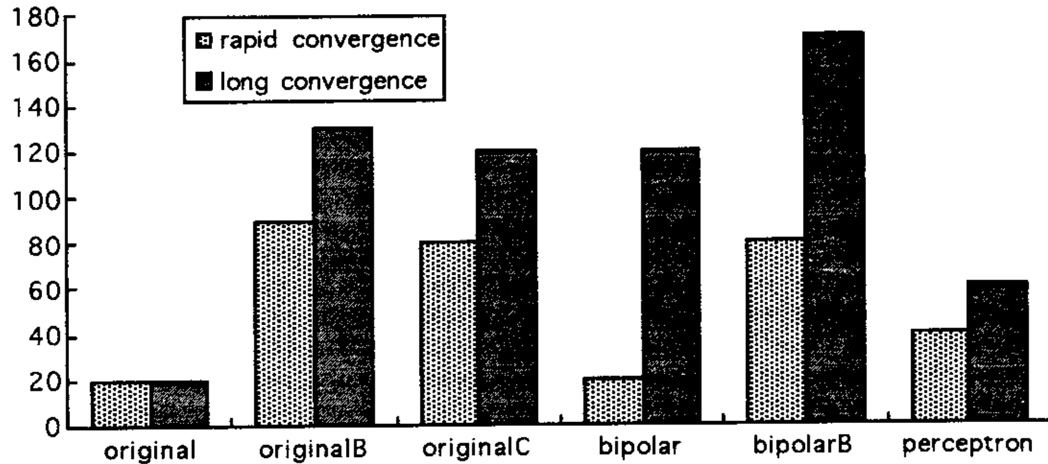


Figure 3.3 Results with the input set *LsTest_50.*

only update their input traces for input 1-bits, this extrapolation can not be made. However, I have observed that the convergence characteristics for these models degrades as the bit density increases.

In light of this, *LbipolarB* can be seen as the best choice for the learning module in general, since it exhibits the most stable convergence behavior throughout the range of bit density settings. In Chapter Four, some further experiments with *LbipolarB* will be reported. *Lbipolar* is perhaps the best choice for $b=0.50$ or lower.

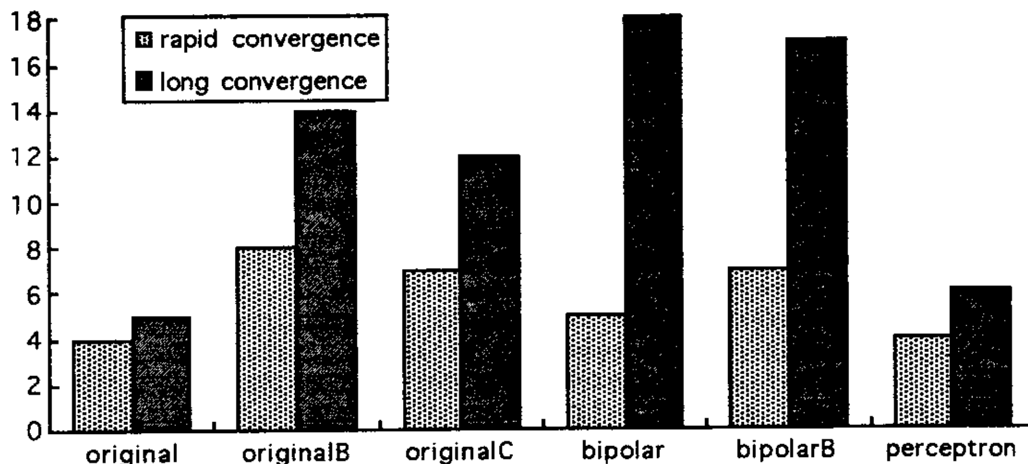


Figure 3.4 Comparative results for six learning modules on the input set *LcTest_50.*

3.7.1 Why the Bipolar Model Shows Better Convergence

The question remains as to why the bipolar implementation is so successful, relative to the dual LTV approach for the three original implementations. For instance, *LoriginalC* and *LbipolarB* are directly comparable, in that they both employ linear update rules and are actively sensitive to both input logic levels.

LbipolarB performs as well as or better than *LoriginalC* in every test, especially in the sparse input case. Perhaps this can be explained by noting that in the dual LTV component models, only one of ***pLTV*** or ***nLTV*** is ever updated at a time. Consequently, each component is trained on a subset of the available information. In cases of sparse input (and output), the ***nLTV*** is updated far more frequently than the ***pLTV***, because the desired output is more often 0 than 1. Since the LTV values are normalized, the ***nLTV*** may not be able to compete if the ***pLTV*** rarely changes. Perhaps a solution in this case might be to use a different normalization threshold for each component. When the input is sparse, the ***nLTV*** might be given a higher activity boundary, allowing it to surpass a stagnant ***pLTV***.

Chapter Four: Some Further Simulations

The simulation results in this chapter were obtained using the full-logic bipolar learning module *LbipolarB*. In each case, tests were performed using an input bit-density of 50% ($b=0.50$), a single sequence of length 100 ($L_s=100$, $L_c=1$), and 5 neurons ($N_n=5$). As in Chapter Three, an input set was generated which includes five trials for each parameter setting. In this case the input set includes corpi for testing with varying numbers of inputs. All tests in this section were conducted using the same input set, *NpTest_50*.

4.1 Testing Convergence Relative to the Number of Inputs

In the first test, I measured the network's convergence based on the number of inputs N_p, given a constant dendritic density of 25% ($D_d=0.25$). Note that as N_p increases, the number of dendritic compartments per neuron N_c will also increase. Alternatively, this experiment can be viewed as testing for convergence as a function of the number of dendritic compartments.

Table 4.1 displays the results of this test. For each input corpus, the network was allowed a maximum of 25 repetitions to train. As expected, convergence improves as N_p and N_c increase.

One reason for this is that the sequence length remains constant ($L_s=100$). As the number of the input bits increases, the number of different possible vectors increases exponentially. Thus, the task of classifying a constant number of vectors becomes

		Repetitions to Convergence (*or correct ratio after 25 reps*)				
N_p	N_c	*1*	*2*	*3*	*4*	*5*
16	64	(.75)	(.71)	(.72)	(.70)	(.70)
32	128	(.996)	(.986)	(.98)	(.97)	(.99)
48	192	21	14	22	20	15
64	256	14	10	12	13	11
80	320	9	11	12	10	9
96	384	8	11	7	9	8

Table 4.1 Convergence results for different numbers of inputs, with constant dendritic density.

progressively easier.

Also, as the number of dendritic compartments increases while L_s remains constant, the network's capacity to distinguish between different states within the input trace values improves.

4.2 Testing Convergence Relative to Dendritic Density

In this experiment, I attempted to test convergence as the dendritic density decreases. Two tests were run, one with $N_p=32$ and the other with $N_p=64$. The results are tabulated in Tables 4.2 and 4.3.

These results are quite interesting. Convergence in each test improves dramatically as the dendritic density decreases from 1 to .25. However, very little change in convergence is observed as D_d is reduced further. In several cases, the performance actually becomes worse with decreased D_d (each column in Tables 4.2 and 4.3 represents

		Repetitions to Convergence (*or correct ratio after 25 reps*)				
D_d	N_c	*1*	*2*	*3*	*4*	*5*
1	32	(.52)	(.57)	(.54)	(.66)	(.56)
.5	64	(.74)	(.64)	(.71)	(.80)	(.71)
.33	96	(.94)	(.83)	(.88)	(.97)	(.90)
.25	128	(.996)	(.98)	(.98)	(.97)	(.99)
.20	160	(.98)	(.98)	(.95)	(.99)	(.98)
.167	192	25	(.95)	(.96)	(.996)	(.998)
.143	223	(.99)	(.97)	(.98)	(.99)	(.98)
.125	256	(.98)	(.98)	(.98)	25	24

Table 4.2 Convergence results for decreasing dendritic density, for a network with $N_p=32$.

		Repetitions to Convergence (*or correct ratio after 25 reps*)				
D_d	N_c	*1*	*2*	*3*	*4*	*5*
1	64	(.66)	(.65)	(.69)	(.67)	(.63)
.5	128	13	12	14	16	18
.33	192	13	9	13	15	12
.25	256	14	10	12	13	11
.20	320	14	12	12	13	15
.167	384	9	10	10	11	11
.143	447	12	10	10	15	11
.125	512	11	12	11	11	13

Table 4.3 Convergence results for decreasing dendritic density, for a network with $N_p=64$.

tests run on the *same* input corpus).

The setting for the shift decay constant might directly affect a network's performance relative to dendritic density. In the above tests, the default shift decay constant of 0.5 was used. After shifting an ITV value 3 times using this setting, an input trace is reduced to only 12.5% of its original value. After being shifted 7 times, a trace is only 0.8% of its former self.

As the dendritic density increases, the number of hidden compartments between each compartment receiving input increases. When the shift decay is as active as in the above tests, it may cause the input trace values to be virtually invisible near the end of a set of hidden compartments.

Perhaps a better experiment would be to vary the shift decay function in addition to the dendritic density. One finding might be that the optimal setting for the shift decay function is dependent on dendritic density.

4.3 Testing Convergence with Duplicate Input Vectors

Up until this point, all tests in this thesis involved input sets with all unique vectors. A model which uses context information in distinguishing its input should be able to correctly classify redundant vectors which occur at different points within the input sequence.

Table 4.4 displays the results for a test in which duplicate vectors were added to the same input corpi at increasing rates of redundancy r_s. A network with $N_p=64$ and $D_d=0.25$

r_s	Repetitions to Convergence				
	1	*2*	*3*	*4*	*5*
0.0	14	10	12	13	11
0.1	13	15	12	11	13
0.2	12	12	12	11	15
0.3	13	14	18	14	14
0.4	15	21	16	14	16
0.5	18	17	25	19	17

Table 4.4 Convergence results for increasing sequence redundancy r_s.

was used.

These results show that the model shows no ill-effects from duplicate vectors for a redundancy ratio of 20% or less. As r_s increases from 30% to 50%, the time to convergence increases gradually.

4.4 Testing Convergence with Noisy Input

It is of interest to test the KATAMIC model's ability to recognize a sequence it has learned in the event of noisy input. As defined in section 2.4, the amount of noise n in the input is the probability that each bit has been changed in value.

An experiment was run in which a network with $N_p=64$ and $D_d=0.25$ was used. Noise was introduced to pre-learned input corpi at a successively increasing rate. Table 4.5 displays the results for this experiment.

It appears in this test that the model is quite sensitive to input noise. Although performance degrades gradually, the output generated is generally 1.5 to 2 times more noisy than the input.

	Correct Ratio after 25 Repetitions				
n	*1*	*2*	*3*	*4*	*5*
.01	.98	.98	.97	.97	.98
.03	.95	.94	.94	.93	.93
.05	.91	.91	.91	.91	.90
.10	.84	.84	.85	.85	.85

Table 4.5 Convergence results for increasing input noise *n*. Each table entry represents an averaged result from five different sub-trials in which random noise was re-introduced to the same input corpus.

4.5 Removing the ITV Wrap-Around During Shifting

Up until this point, the shifting of input trace values during the KATAMIC cycle has been implemented to wrap-around from one end of the chain of dendritic compartments to the other. It would be a useful simplification if the wrap-around could be removed, since this would substantially simplify the model's potential hardware implementation.

wrap around	Repetitions to Convergence				
	1	*2*	*3*	*4*	*5*
on	14	10	12	13	11
off	11	12	10	12	11

Table 4.6 Convergence characteristics for a network using N_p=*64* and D_d=*0.25*, with and without the use of wrap-around during the ITV shift operation.

Table 4.6 reports a simple experiment in which the ITV wrap-around was turned on and off, with N_p=*64* and D_d=*0.25*. Over five trials, it appears in this case that there is no drop in performance when the wrap-around is removed (in fact, convergence is slightly improved is some instances). This is not a surprising result, since only 1 of 256 input trace values is lost during each cycle. While this is only a preliminary experiment, it appears that removing the shift wrap-around can be a valid simplification.

Chapter Five: On Scalability and Hardware Implementation

One of the nice things about the KATAMIC model is its relatively simple connectivity. Since all neurons receive the same primary input (or a weighted version of the same input, when spatial decay is used), there is no exponential growth in the number of signals that must crossover each other as the size of the network increases. This useful feature might make KATAMIC nets an attractive choice for hardware implementation.

This is not true for models based on multi-layered feed forward networks, such as back-propagation type recurrent nets, in which each node is fully connected to the nodes in subsequent levels [1]. This means that the number of connections from one node in a level A to a node in a higher level B must crossover an exponentially increasing number of signals as the number of nodes per layer increases.

The need for adding nodes in the hidden layer of feed forward type networks might be for improving storage capacity. In KATAMIC nets, the number of dendritic compartments can be increased to augment capacity. This can be implemented such that the degree of signal crossover does not increase.

While each neuron must be capable of calculating an activation function (Euclidean dot product) based on the contents of each of its dendritic compartments, this can be implemented as a serial process, in which the computation begins with the most distal dendritic compartment, and progresses towards the neuron's output port. In this case, each compartment would need to calculate the product of its ITV and LTV components and add it to the accumulated total of the previous compartments. Structure already in place for shifting and decaying ITV values could be used for this purpose. However, this would cause a linear increase in computation time, relative to the growth in the number of compartments per neuron.[7]

[7]A simple perceptron-like neuron used in the layers of a feed-forward network must still be able to handle many inputs within a dendritic structure, however. Any implementation will show some reduction

Although each KATAMIC neuron is considerably more complex than are the nodes for many other models, I view the KATAMIC network as relatively simple, since its scalability in terms of signal crossover and dendritic density is non-exponential. It can perhaps be speculated that because of its single layer topology, the KATAMIC neuron has a limited ability to perform any arbitrary classification of its temporal input, in a manner similar to the problems of the single-layer perceptron in recognizing the exclusive-or function [6]. While this issue remains to be investigated, I would argue that the KATAMIC model is clearly useful over a wide range of input spaces.

---

in speed as the number of inputs to a node increases. The KATAMIC neuron compounds this by adding to the computation with hidden compartments.

Chapter Six: Concluding Remarks

6.1 The Virtues of Simplicity

A well known saying in engineering circles is "simple is better." This is true, since the cost of design for complex systems is directly affected by the ability of those who build and maintain them to understand and optimize performance. My work with the KATAMIC model was originally motivated by a desire to simplify for simplifications's sake. In doing so, my understanding of the KATAMIC learning process was enhanced, allowing me to improve its learning characteristics.

The non-sigmoidal bipolar learning model represents a substantial refinement of Nenov's original description. In addition, the results included in this thesis demonstrate that this modification is not only valid, but quite desirable in terms of performance.

While I have concentrated on the essential learning properties within the dendritic structure of the KATAMIC neuron, any results should be quite extendible to Nenov's fuller version of the model, which included structure for recognition and feedback.

Important findings of this thesis include:

> (1) The learning mechanism can be simplified to require half the storage for learned trace values, with a significant reduction in the number of computations needed per cycle.[8] More importantly, this modification greatly improves the convergence behavior of the model.
>
> (2) The spatial decay function between neurons along the input parallel fibers can be replaced in the most general case by a flat constant function. This has the effect of rendering each neuron independent from the rest of the network.
>
> (3) Several further simplifications have proven to be valid modifications which offer significant speedup in computation, including the use of a linear update function during training (as opposed to sigmoidal).
>
> (4) The format of the input can be switched to have values of -1 and 1 (instead of 0 and 1), allowing the model to be successful in a broader range

[8] Michael McNally first successfully demonstrated this simplification [5].

of input 1-bit densities (previously, the model was only successful in cases of relatively sparsely populated input vectors).

(5) Altogether, simulation results show that improvements to the model can result in a speedup of 4 to 1, and an increase in learning capacity of six times.

6.2 De-Constructing the Neuroscience Perspective

The original design of the KATAMIC network was rooted very closely to known neurological structure. In fact, Valeriy Nenov completed his Ph.D. in neuroscience before continuing on for that same degree in computer science [7].

The KATAMIC framework is modeled after the relatively regular structure of the human cerebellar cortex. In the cerebellum, the largest and most important neurons are called Purkinje cells. Purkinje cells are organized in one-dimensional rows, while smaller granular cells reside beneath them. The axons of these granular cells extend upward to the level of the Purkinje dendrites, and then bifurcate in order to extend in either direction along the axis of the Purkinje cell rows. These fibers are called *parallel fibers* and connect to the dendritic arborization of each Purkinje cell in their path. Thus, each Purkinje cell in a row receives similar input, although there may be a time-delay along the parallel fibers.

Nenov's *predictrons* are the KATAMIC analog to the cerebellar Purkinje cell. The additional structure in Nenov's original description was also motivated by cerebellar structure. Although Nenov makes no claims that the KATAMIC network should be viewed as a model of the cerebellum, it does exhibit some similar behavior. The cerebellum is considered to be a regulator of motor function. It does not initiate motor control, but it does smooth and facilitate the process. Likewise, the full KATAMIC model was designed to learn to predict its input and regularize it on output. Since it is able to selectively route its output to its input, the KATAMIC neuron can effectively run by itself after having received an initial stimulus.

It is an ongoing debate within the cognitive science community whether it is more important to replicate and simulate exactly existing structure within the human nervous system, or whether to build models based solely on their perceived computational merits. Clearly, I think the answer lies in between. Understanding can only be enhanced by greater interaction between research in these related fields of study. This is not an easy proposition, however, since historically there has been so little overlap in the educational backgrounds of researchers in computer science and neuroscience.

There is a need for bi-directional feedback between the computational sciences and the neurological and cognitive sciences. Nenov bridged the gap, and in the process started the useful study of a promising temporal recognition mechanism.

It has been enlightening for me to learn introductory neuroanatomy, having come from a computer science background. In so doing, my understanding of the KATAMIC model has been enhanced, while at the same time I can now fully appreciate how far from its neurological beginnings the model has come. As I have attempted to simplify and improve the model, I have also removed much of its original neuroscientific flavor. It is perhaps ironic that as I came to appreciate the neuroscience perspective, I sought only to extract, isolate and formalize the most computationally expedient pieces of the puzzle. While my results appear useful and interesting, they are still relatively inconsequential when compared against the tremendous cognitive power of the "real thing."

Appendix A: Changes to KATAMIC Terminology

In describing the KATAMIC learning module, I have taken the liberty of renaming some of the features, as compared to their description by Valeriy Nenov [9]. In addition, I have only concentrated on part of the original model.

Nenov's KATAMIC net has three basic processing units. Of these, the *predictron* is the most interesting, in that it is where the essential temporal learning process occurs. As noted in section 3.1.2, Nenov thought of this unit as a predictor of its next input. However, I have generalized the description to the case where the network can be trained to learn any arbitrary sequence (of which predicting the next input is a special case). As a consequence, I have not retained the term predictron. Instead, I view it as the fundamental neural element around which all KATAMIC networks are based. Thus I have referred to Nenov's predictron simply as a *KATAMIC neuron* (or *neural processing element*).

In addition, I have also changed terminology regarding the internal structure of dendritic compartments within Nenov's predictron. He referred to the memory components as short term memory (STM) and long term memory (LTM, which is made up of pLTM and nLTM). I have changed these to the input trace value (ITV) and the learned trace value (LTV), respectively.[9] I feel these to be intuitive changes which add to the descriptive nature of the KATAMIC definition. The original terms do not highlight the essential function of these components, which is to store and temporally encode the input over time as a *trace*. I view this as one of the unique aspects of the KATAMIC framework.

I have also condensed and simplified the outline of steps for executing a KATAMIC cycle. Where I refer to the *input* step, Nenov described the process as the three steps: *get input*, *inject STM*, and *update STM*. Nenov also refers to *LTM resource maintenance* --

---

[9]Actually, Valeriy Nenov suggested the use of ITV and LTV as opposed to ITR and LTR, which I was using initially.

forgetting. I have renamed this step *normalization.* I have also changed the step *temporal encoding* to *shifting*, and the step *predict next input* to *output*.

In addition to the predictron, Nenov also described the *recognitron* and the *bi-stable switch* (BSS). Recognitrons are intended to recognize when one or more predictrons within a neighborhood have produced the correct output. There is one recognitron per predictron, which controls a BSS. The BSS is essentially a multiplexer, whose output becomes the parallel fiber connected to the seed compartment of the predictron with which it is associated. Based on the current state of the recognitron, the BSS switches its source between the current external input and the internal input prediction (which is the last output of the predictron). If the recognitron asserts that recognition has occurred (to within a threshold), the BSS will continue to use the external input source. If the recognition has failed, however, the BSS will use the current output of the prediction (which is possibly erroneous). The BSS uses an exponential delay to prevent it from immediately switching back to the external input source when recognition has been reasserted. This has the effect of allowing a predictron to "run on its own" for a period of time, if it comes to disagree with the current input. Nenov's KATAMIC cycle had steps for using the recognitron and BSS: *attempt sequence recognition* and *generate next input.*

While I have not addressed the behavior of this added structure, I do feel that there are open questions left for further research. It appears that the recognitron-BSS combination could serve as a regulator of the output, in the event of noisy input. This is similar to the function of the cerebellar cortex, which is thought to smooth control of motor output within the mammalian nervous system.

Appendix B: Overview Ksim

The experiments reported in this thesis were conducted using the simulation environment Ksim. Ksim has evolved to become a versatile KATAMIC simulator. I have designed it in a modular way in order to make it easy to add features and simulation parameters.

Ksim was developed using the object-oriented programming language *C++*, and runs in a UNIX environment. The different KATAMIC learning modules are implemented as data objects, and all use a common interface to the Ksim execution engine.

Ksim is built around a command interpreter. It is also able to run in batch mode from a file of commands, making it easier to run an exhaustive set of repetitive experiments.

Listed in this appendix is a complete set of Ksim commands, followed by a header file for writing learning modules.

B.1 List of Commands for Ksim

All Ksim commands are one or two characters (subsequent characters are ignored). Commands are case sensitive. The commands are divided into seven categories: *main*, *batch*, *configuration*, *input*, *learn*, *routing* and *run*.

B.1.1 Main Commands

h {c | i | l | ru | ro | ba | m}
: Help for the seven different command categories.

v {<var> | c | i | l | ru | ro | m}
: Display the value of a specific scalar variable, or all the variables from one of the categories *configuration*, *input*, *learn*, *routing*, *run* or *main*.

V (<var>) {idx1 {idx2 {idx3}}}
: Display the contents of a vector or matrix variable, within an optional index range.

VM (<var> {idx1 {idx2 {idx3}}}

Same as V, but displays in Mathematica format.

VF (<var> {<filename> {idx1 {idx2 {idx3}}}}
Same as V, but writes in Mathematica format to a file.

sa (c | i | l | ru | ro | all) {<filename>}
Save the current simulation state information to a file. One of the five (or all five) states will be written: *configuration, input, learn, routing* or *run.*

re (c | i | l | ru | ro | all) {<filename>}
Restore the current simulation state information from a file. One of the five (or all five) states will be read: *configuration, input, learn, routing* or *run.*

os {<filename>}
Set output stream, which will duplicate all information printed to standard output.

q Quit Ksim.

B.1.2 Batch Commands

ba {<filename>}
Do commands from a batch file.

se <strNum> <string>
Set value for 1 of 20 string variables. Once set, a string can be echoed using '$<strNum>'.

se <strNum> %<var>
Set the value of a string using the value of an environment variable.

se <strNum> %pr <promptString>
Ask the user to enter input, after prompting with promptString.

sv Show the current value for all string variables.

pr <prompt>
Echo prompt to screen

st Set the date_time stamp to the current date and time.

dt Show the current date_time stamp, without resetting it.

ao Set/reset the askOverWrite flag, which determines whether to ask the user before overwriting existing files.

fe Set the fileError flag (usually for resetting after error). Resetting this will turn off fileReadError and fileWriteError. Any executing batch file will cause Ksim to quit if fileError becomes true.

fr Set the fileReadError flag. Setting this will turn on fileError.

fw Set the fileWriteError flag. Setting this will turn on fileError.

B.1.3 Configuration Commands

Nn {<val>}
Set value for Nn (the number of neurons in a network).

Nc {<val>}
Set value for Nc (the number of dendritic compartments per neuron).

Np {<val>}
Set value for Np (the number of input parallel fibers).

Dd {<val>}
Set value for Dd (the dendritic density).

KF {<funcNum>}
Set KFunction (the mapping function for the parallel fiber distribution vector kVec).

SF {<funcNum>}
Set SFunction (the mapping function for the seed distribution vector sVec).

dd {<funcNum>}
Set ddFunction (the function for determining the spatial decay, which initializes the set of decay weights, stored in ddMatrix).

v (c | Nc | Nn | Np | Dd | Ds | KF | SF | dd)
Display a configuration scalar variable ('c' shows all configuration variables).

V|VM|VF (k | s | dd) {idx1 {idx2}}
Display kVec, sVec or ddMatrix.

B.1.4 Input Commands

bd {<val>}
Set the input bit density variable.

sb {<val>}
Turn on/off strict bit density flag.

si {<val>}
Turn on/off similarity checking flag.

ms {<val>}
Set maximum similarity to be allowed in input (if si turned on).

Ls {<val>}
Set the sequence length.

Lc {val}
Set the corpus length.

rs {val}
Set the sequence redundancy rate.

rc {val}
Set the corpus redundancy rate (only if rs=0).

ii Re-generate a new input corpus (this is done automatically if relevant input parameters are changed).

di Add duplicates to current input based on rs or rc.

v (i | bd | Ls | Lc | rs | rc)
Show the value of an input variable ('i' shows all input variables).

(V|VM|VF) (i|_i|I|C) {idx1 {idx2 {idx3}}}
Display current input corpus, sequence or vector (_i is the internal input vector).

B.1.5 Learn Module Commands

lM {<moduleNumber>}
Choose one of the installed learning modules.

iI Re-initialize ITV.

iL Re-initialize LTV.

v <var>
Show learn module variable(s). These can vary for different modules ('h l' will cause the current learn module to display its own help screen).

V (IT|LT) {idx1 {idx2}}
Display ITV or LTV state.

The following commands are active with the bipolar learning module:

Ii Set ITVinit.

Li Set LTVinit.

Ir Set ITV update rate.

Lr Set LTV learn rate.

oT Set outThreshold.

Lt Set LTVthreshold (the amount of activity per dendritic compartment to be used for normalization).

nT Set normThreshold (default is Nc*LTVthreshold).

sD Set shiftDecay rate.

B.1.6 Run Commands

r {<seq> {<vec>}}
: Reset the network and set the input corpus to the sequence and vector pointed to by seq (default=0) and vec (default=0).

g (s | c | S | C) {<num>}
: Run num (default=1) steps, cycles, sequences or corpus reps.

g U {<mThresh> {<sThresh> {<maxReps> {<plusReps>}}}}
: Run corpus until: matches>=mThresh (default=1), spurious<=sThresh (default=0), or until maxReps (default=100). Run for plusReps after condition met (default=0).

g u {<cThresh> {<maxReps> {<plusReps>}}}
: Run corpus until: correct>=cThresh (default=1) or until maxReps (default=100). Run for plusReps after condition met (default=0).

cy Display current cycle step order.

co {<orderNum}
: Set cycle order; will prompt if orderNum not specified.

ls {<fileName}
: Set stream for logging matches/spurious/correct will prompt if fileName not specified; default is NULL(no logging).

le {<val>}
: Turn on/off learning flag.

no {<val>}
: Turn on/off normalizing flag (turned off if learning is turned off).

sh {<vat>}
: Turn on/off shifting flag.

wr {<val>}
: Turn on/off wrapAround during shifting of ITV.

sd {<val>}
: Turn on/off spatial decay flag.

sI {<val>}
: Turn on/off single ITV flag (can only be on if spatial decay flag is off).

B.1.7 Routing Commands

ti Set teach initializer function (can be set to random, or to predict next input).

ri Set input/output routing function (simple pass through, or introduced noise).

V (t | T | Ct | o | _o) {idx1 {idx2}}
Display teach/output matrices.

v (ro | ti | ri)
Show value of routing variable(s).

B.2 Template for Learning Module Header File

Listed here is the header file "learnbase.h". All learning modules must be defined as derived classes of learnbase.

```
#ifndef _learnbase_h
#define _learnbase_h

#include <String.h>
#include "mystream.h"
#include "int.AVec.h"
#include "doubleAVec.Vec.h"
#include "commands.h"

//#included in "run.h"
extern short useSingleITV,useSpatialDecay;

//#included in "config.h"
extern doubleAVecVec *ddMatrix;

class learnbase {
                String nm;
        public:
                learnbase();
                learnbase(int Nn,int Nc);
                virtual ~learnbase();
                void setName(String& nmStr){nm=nmStr;}
                String& name(){return(nm);}

                virtual void resize2(int Nn,int Nc)=0;
                virtual void uITV(intAVec& iVec)=0;
                virtual void uLTV(intAVec& lVec)=0;
                virtual void shiftITV()=0;
                virtual void shiftITVwrap()=0;
                virtual void normLTV(intAVec& lVec)=0;
```

```
                    virtual void initITV()=0;
                    virtual void initLTV()=0;
                    virtual void output(intAVec **o)=0;
                    virtual short saveLearnState(fstream& stateFile)=0;
                    virtual short restoreLearnState(fstream& stateFile)=0;
                    virtual void displayITV(fstream& dispStream=coutStream,
                            int start=0,int end=-1);
                    virtual void displayLTV(fstream& dispStream=coutStream,
                            int start=0,int end=-1);
                    virtual void displayMathematicaITV(String& varStr,
                            fstream& dispStream=coutStream,
                            int start=0,int end=-1);
                    virtual void displayMathematicaLTV(String& varStr,
                            fstream& dispStream=coutStream,
                            int start=0,int end=-1);
                    virtual short setParams();
                    virtual void showParams(short inVal=0);
                    virtual void paramHelp();
};

inline learnbase::learnbase(){nm="learnbase";}
inline learnbase::learnbase(int Nc,int Nn){nm="learnbase";}
inline learnbase::~learnbase(){}

#endif
```

References

[1] Elman, J.L., "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179-211, (1990).

[2] Hopfield, J.J., "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences, USA [Biophysics]*, vol. 79, pp. 2554-2558, (1982).

[3] Jordan, M.I., "Attractor dynamics and parallelism in a connectionist sequential machine," *Proceedings of the 8th Conference of the Cognitive Science Society*, pp. 531-546, Lawrence Erlbaum Associates, Hillsdale, NJ (1986).

[4] Kohonen, T., *Self-organization and associative memory*, Springer-Verlag, Berlin (1984).

[5] McNally, M., personal communication, (December 1992).

[6] Minsky, M. and Papert, S., *Perceptrons: An introduction to computational geometry*, MIT Press, Cambridge, MA (1969), (Reissued in an Expanded Edition, 1988).

[7] Nenov, V.I., "Metabolic and Electrophysiologic Manifestations of Recognition Memory in Humans," Ph.D. Dissertation, Neuroscience Program, University of California, Los Angeles, CA (1989).

[8] Nenov, V.I., "Rapid learning of pattern sequences: A novel network model," *Proceedings of the International Neural Networks Conference (INNC-90)*, Paris, (abstract only), (1990).

[9] Nenov, V.I., "Perceptually Grounded Language Acquisition: A Neural/Procedural Hybrid Model," Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA (1991).

[10] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," pp 318-362 in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol 1, ed. D.E. Rumelhart and J. L.McClelland, MIT Press, Cambridge, MA, (1986).

[11] Stroustrup, B., *The C++ Programming Language*, 2nd ed. Addison-Wesley, Menlo Park, CA (1991).